NAVAL POSTGRADUATE SCHOOL

Monterey, California

Underwater Multi-dimensional Path Planning for the
Naval Postgraduate School Autonomous Underwater Vehicle II

by

Joseph Bonsignore, Jr.

September, 1991

Thesis Advisor: Yuh-jeng Lee

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS(52)	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Underwater Multi-dimensional Path Planning for the Naval Postgraduate School Autonomous Underwater Vehicle II

12. PERSONAL AUTHOR(S)
Joseph Bonsignore, Jr.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 9/89 TO 9/91	September 1991	151

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Path planning, Path replanning, Tendril search, Wavefront search, Real-time A* search, Autonomous Underwater Vehicle, AUV, Ada in ArtificialIntelligence.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Traditionally path planning software has been developed in LISP or C. With the recent government mandate for the use of Ada, this thesis seeks to demonstrate the feasibility of using Ada for both path preplanning and real-time path replanning. Land vehicle path planning can be accomplished with two horizontal components. However, for autonomous underwater vehicles, the two horizontal components and a vertical component are required. Memory and computational speed restrictions dictate that special processing of the search space be conducted to optimize the time-space trade-off. In this research, a four dimensional array of nodes (two horizontal components, one vertical component and one orientation component) is used to represent the search space. By use of an orientation component, the number of nodes that can be legally moved to is limited, in effect pruning the search space. Three search methods were investigated: the Tendril search, the Direction search and the Real-time A* search. The Tendril search is a wavefront, breadth-first search. The Direction search uses a vector field for path planning. The Real-time A* search uses the Tendril search to a specified search depth then applies a heuristic to determine the best path to expand upon.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Yuh-jeng Lee	22b. TELEPHONE (Include Area Code): (408) 646-2361	22c. OFFICE SYMBOL: CS/ (52)

Approved for public release; distribution is unlimited

Underwater Multi-dimensional Path Planning
for the Naval Postgraduate School
Autonomous Underwater Vehicle II

by

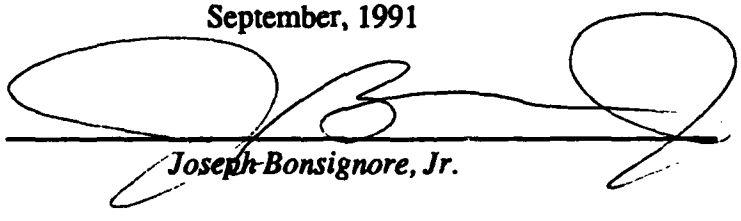
Joseph Bonsignore, Jr.
Captain, United States Marine Corps
B.S., The Virginia Military Institute, 1979

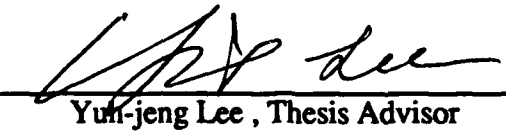
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

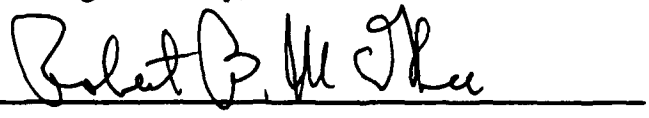
from the

NAVAL POSTGRADUATE SCHOOL
September, 1991

Author: 
Joseph Bonsignore, Jr.

Approved By: 
Yuh-jeng Lee , Thesis Advisor


Leigh Bradbury, Second Reader


Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

Traditionally path planning software has been developed in LISP or C. With the recent government mandate for the use of Ada, this thesis seeks to demonstrate the feasibility of using Ada for both path preplanning and real-time path replanning. Land vehicle path planning can be accomplished with two horizontal components. However, for autonomous underwater vehicles, the two horizontal components and a vertical component are required. Memory and computational speed restrictions dictate that special processing of the search space be conducted to optimize the time-space trade-off. In this research, a four dimensional array of nodes (two horizontal components, one vertical component and one orientation component) is used to represent the search space. By use of an orientation component, the number of nodes that can be legally moved to is limited, in effect pruning the search space. Three search methods were investigated: the Tendril search, the Direction search and the Real-time A* search. The Tendril search is a wavefront, breadth-first search. The Direction search uses a vector field for path planning. The Real-time A* search uses the Tendril search to a specified search depth then applies a heuristic to determine the best path to expand upon.

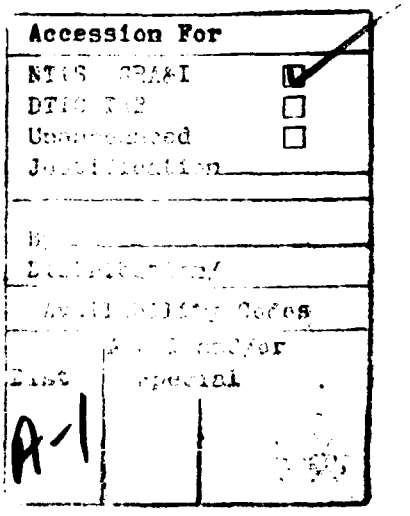


TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Autonomous underwater vehicle (AUV) research continues to grow as more applications are devised. From industry and scientific research to military applications, AUV technology has generated great interest. Currently, there are nearly 30 different organizations researching AUV technology, of which 18 are government funded [Busby and Vadus, 90]. This indicates the strong interest the government has in this technology.

The benefits provided by unmanned autonomous vehicles are many. They provide a means to accomplish missions which are considered too dangerous for human involvement [Cloutier 90]. "Progress is aimed toward minimizing need for man's physical presence, intervention underwater" [Busby and Vadus 90]. Underwater vehicles can be categorized as either tethered or autonomous [Rogers 89]. In contrast to a remotely operated vehicle (ROV), an AUV is not restricted by an umbilical which can hinder task performance in some cases.

Due to the AUV's nature, mission planning and execution are very complex problems to solve. Accurate world models must be made and complex path planning performed prior to mission execution. During task performance, continued evaluation of the many aspects of the mission must be performed. If necessary, adjustment or replanning must be conducted to insure successful mission completion or a decision to abort.

A. OBJECTIVES

This thesis intends to focus on path planning and replanning using the Ada programming language [Healey 90]. Several objectives are listed below:

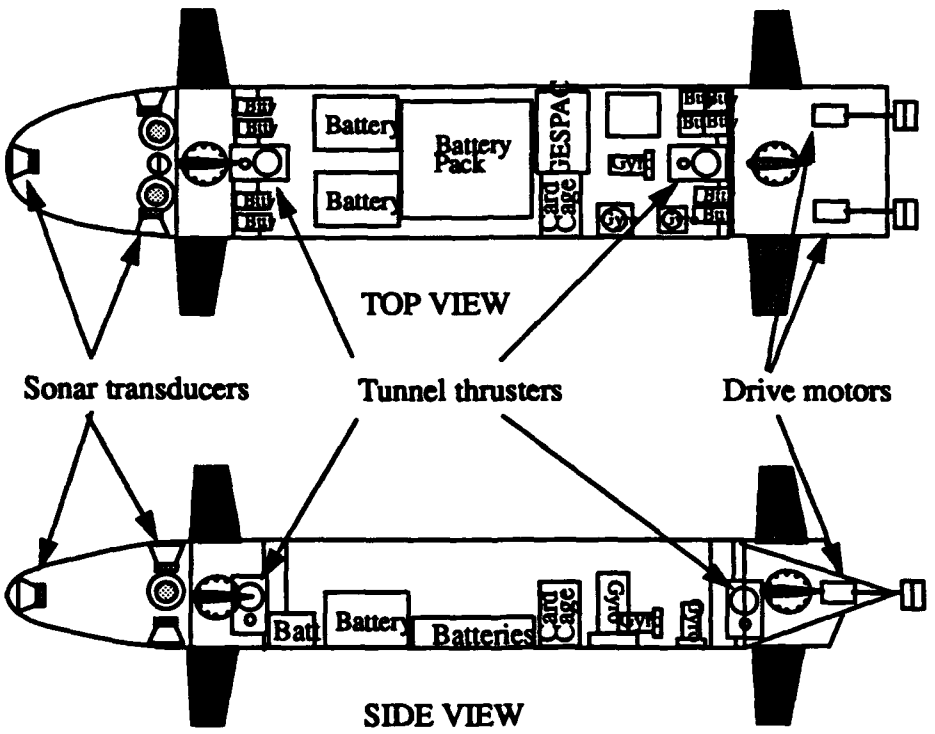
1. Implement a multi-dimensional Tendril search in Ada.
2. Investigate the feasibility of waypoint utilization.
3. Implement a Real-time A* (RTA*) path replanner in Ada.
4. Investigate the feasibility of a vector field method of path planning.

5. Examine the feasibility of utilizing Ada reusable modules.

B. BACKGROUND

1. Naval Postgraduate School AUV II (NPS AUV II)

The basic component layout of the NPS AUV II is illustrated in Figure 1-1. It is of aluminum box construction with a 16" beam, 10" height, 92" length and displaces 390 pounds [Cloutier 90]. It uses eight independent control surfaces, four tunnel thrusters and two main screws, and has a top speed of two knots (three feet per second) with a 20 feet



NPS AUV II

Figure 1-1

turning diameter. Power is provided by on-board lead-acid batteries with an approximate two and one half hour operating time [Floyd 91]. Current on-board systems include a GESPAC MPU 20HF board with Motorola 68020 and 68882 processors. OS - 9 was chosen as the operating system for its multi-tasking capabilities. As recommended by Bihari, a full GESPAC suite is expected to be used for its suitability to AUV applications [Bihari 90]. Appendix A and B are Data Flow Diagrams (DFD) and Software Heirachy for this project.

2. Mission Planning Expert System

The mission planning Expert System (MPES) is hosted on a Symbolics 3675 LISP machine. Using the KEE expert system shell, it has four major components: the Mission Receiver, Mission Planner, Mission Constructor and Mission Executor. The Mission Receiver acts as the interface agent for user input. This information is passed to the Mission Planner which decides which path planning algorithm is best suited for the user supplied circumstances. Using various search technique including A*, and best-first search, the Mission Constructor does the actual path planning. The Mission Executor interfaces with the AUV/simulator and provides the appropriate mission data for execution. Figure 1-2 illustrates the MPES structure. [Ong 90]

3. NPS AUV Simulator

The NPS AUV II simulator contains a full set of submarine motion and hydrodynamics equations providing accurate, real-time simulation. Implemented on a SGI IRIS 4D/240 GTX graphics workstation, it displays a detailed underwater mapping of the Monterey Bay. Variable terrain resolution is used automatically to allow real-time operations. Its development is a joint effort between the computer science and mechanical engineering departments at the Naval Postgraduate School. [Jurewicz 90]

C. THESIS ORGANIZATION

Chapter II reviews previous and current work in AUV technology. A detailed look at various path planning techniques is provided.

Chapter III introduces the Tendril search. The original two dimensional and the expanded four dimensional version in LISP are examined. Several learning points from the task of translating the LISP program to Ada are reviewed. Limiting features of the four dimensional Tendril search are presented as well as a look into the feasibility of the Tendril bidirection search (TBS).

Chapter IV presents a vector field approach to path planning with a discussion of advantages and disadvantages.

Chapter V presents a real-time path r 'anner using the Real-time A* Search (RTA*). Many questions are posed and plausible justification for the use of the RTA* is presented.

Chapter VI provides conclusions and recommendations for further research. Heavy emphasis is placed on the recommendations, which provide good insight to the perceived goals of the NPS AUV research.

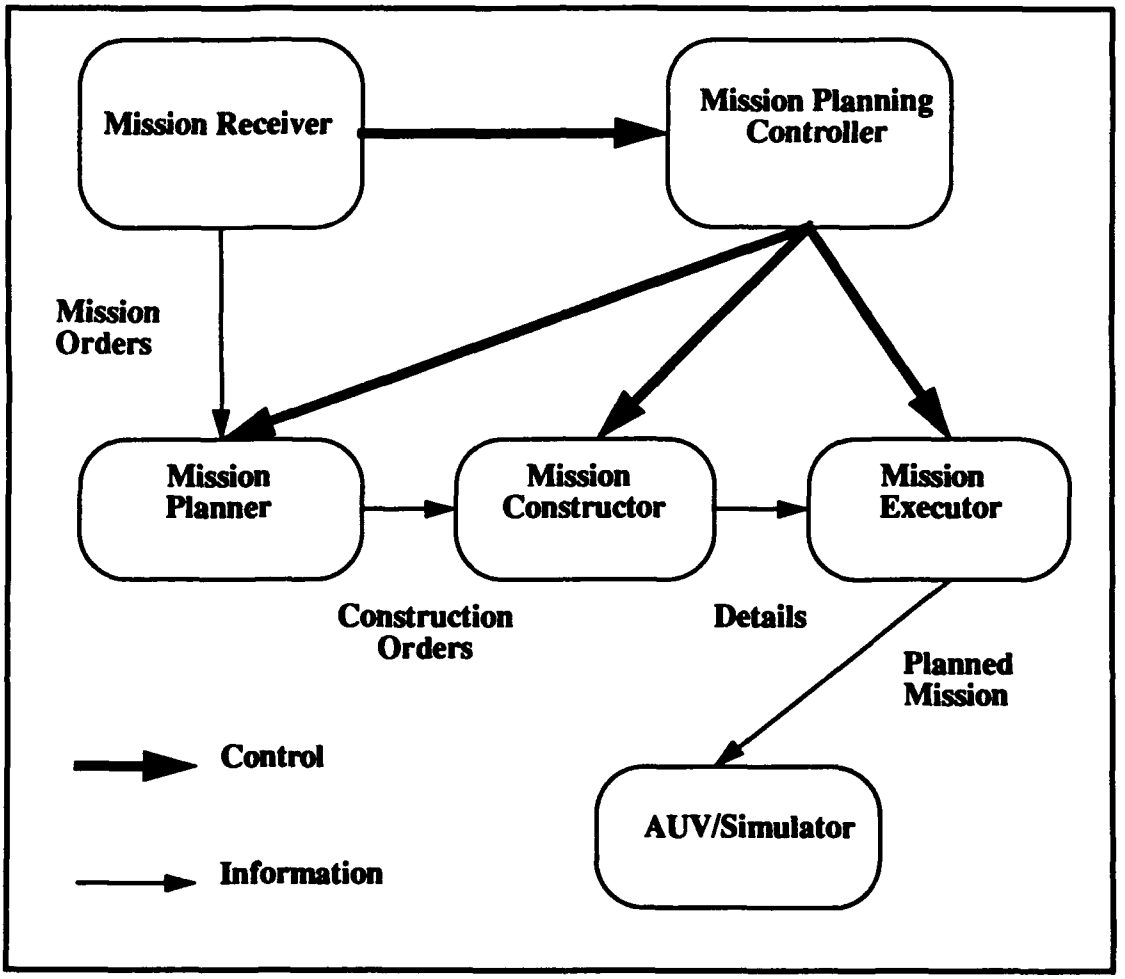


Figure 1-2 Mission Planner Program Diagram

II. AUTONOMOUS UNDERWATER VEHICLE RESEARCH

A. VEHICLE ARCHITECTURE

1. Texas A&M

Texas A & M University has made major contributions to AUV technology conducting feasibility research. Reliability and fault tolerance were the primary concerns, and no vehicle construction was intended. To accommodate this, 16 Sun Sparc stations, fully distributed, were used (fully loosely coupled). [Cloutier 90]

Programmed in the C language, it incorporated nine embedded knowledge based systems (KBS). Some important findings resulted from this research. The use of a "watch team" knowledge base (KB) was overly centralized. This KB simulated the tasks and duties performed by a human team aboard Navy submarines. By its centralized nature, results and decisions were predictable, however, flexibility was reduced. There is a trade-off between flexibility and predictability which must be closely considered [Cloutier 90].

2. Naval Ocean Systems Center (NOSC)

NOSC pioneered AUV research in the 1980's after extended involvement with remotely operated vehicle (ROV) research since the 1960's. NOSC has several on-going research efforts. Advanced unmanned search system (AUSS) was developed for search and survey. The free-swimming mine neutralization vehicle (FSMNV) is also under development. [Busby and Vadus 90]

3. Massachusetts Institute of Technology (MIT)

The Massachusetts Institute of Technology developed Sea Squirt with funding from the National Oceanic & Atmospheric Administration and in cooperation with Draper Labs. It is a light weight, low-cost AUV primarily used as a test platform for intelligent algorithms. The research goal is to make a vehicle capable of operations in an adaptive

manner with respect to its environment in an uncharted area [Busby and Vadus 90]. Onboard systems include a GESPAC MPU - 20 board with a Motorola 68020 cpu and runs OS - 9, like the NPS AUV II [Bellingham 90].

4. Defense Advance Research Projects Agency

The Defense Advance Research Projects Agency (DARPA) has been actively involved in AUV research. It has provided about 90% of the research funding throughout the research community and industry. As early as 1988, DARPA initiated projects with Draper Laboratory and Martin Marietta. These projects focused on the research and development of two vehicles and intelligence task research. Martin Marietta also planned to develop a "hard time" aspect to navigation that entailed planning a path where arrival times at specific way points are known before hand and met during execution. In 1989 DARPA in conjunction with Lockheed Missiles and Space Company began development of an autonomous mine avoidance vehicle. [Busby and Vadus 1990]

5. International Submarine Engineering

Over the years, International Submarine Engineering, Ltd. (ISE) has developed several unmanned underwater vehicles (UUV). Its DOLPHIN was a diesel powered vehicle designed for offshore hydrographic mapping. The interesting aspect of this vehicle is the use of GESPAC components. After an extensive market survey, GESPAC was chosen for its price, performance, size, ruggedness and availability. [Zheng et. al. 90]

B. PATH PLANNING

1. General Path Planning

The ultimate goal of a path planner is to derive a continuous set of free space points from the starting position to the goal position [Latombe 91]. For control of autonomous vehicles this can be done at various levels of resolution. A route planner is used at low resolution and a path planner provides a more specific solution to the path planning problem. In essence, the path planner provides a detailed path for the various path

segments generated by the route planner [Ong 90]. This thesis concentrates on the path planning resolution.

Path planning is a well researched topic with many search techniques being used in various research efforts. Most commonly used methods are: Breadth-first, A*, Hill climbing, Depth-first and Best-first. Some of these will be presented later in this chapter.

Important to note is the complexity of the search methods. As search space increases, some techniques, especially exhaustive methods, tend to be restrictive either due to memory or time requirements. Branching factor is an important aspect to consider. In a two dimensional search space there may be as many as eight adjacent nodes that the vehicle can legally move to as illustrated in Figure 2-1. With the addition of a third dimension these legal moves increase to 26 as illustrated in Figure 2-2. Methods are required to sufficiently reduce the branching factor to make these techniques viable. [Ong 90]

2. Fast, Three-dimensional, Collision-free Motion Planning

a. General Description

Implemented in a nodal search space representation, this method successively divides the search space into homogeneous octrees or to a set resolution limit (section b below provides more detail on octrees). The less node division that is necessary, the easier for the path planner to process the search space. It is evident that planning a path in a search space with a small number of nodes is easier than to do so in a search space with a large number of nodes. [Herman 86]

b. World Representation

Upon initialization, the world is represented as a single node. If the node is not homogeneous (either wholly obstacle or free space) it is divided into eight children nodes. Each new node is evaluated for homogeneity and if necessary divided further. This process continues until all nodes are either wholly obstacles or free space, or the resolution limit is reached. A tree structure is formed with the original node as the parent and the resulting octree nodes as the children.

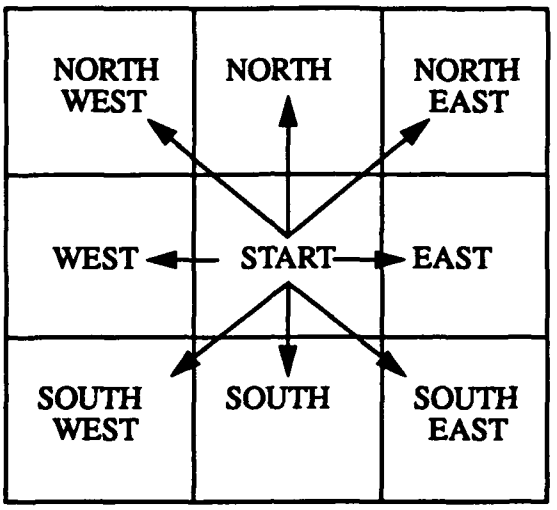


Figure 2-1 Two Dimensional Legal Moves

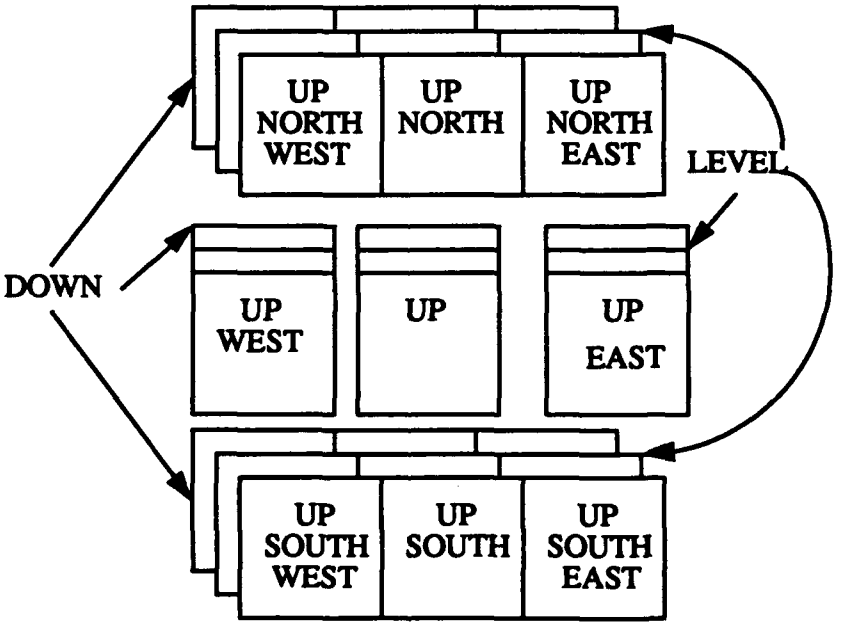


Figure 2-2 Three Dimensional Legal Moves

c. Search Techniques

Several search methods were incorporated in this method, to optimize time and memory use:

1. Hypothesis and Test,
2. Hill Climbing,
3. A*.

The process started with the hypothesis and test method and switched to the hill climbing method. When local maxima were encountered the A* process was used to overcome them. Once past the local maxima, the program continued with the hill climbing process.

d. Conclusions

It is important to note that this method may not find the minimal cost path and only finds an adequate cost path. This may or may not be sufficient for some path planning needs.

Another consideration is the use of several methods for path determination. The advantage of using multiple methods is to avoid "traps" such as local maxima that could stop a search from finding a path to the goal where one exists. Other individual methods may not encounter local maxima "traps" but may be time restrictive. Thus for timing considerations a less exhaustive method is used and the "traps" must be considered. By making this time trade-off, other methods are required where one method may fail. Although acceptable, using multiple methods adds to the complexity of the problem and appears to provide little overall advantage in computational speed.

3. Bidirectional Staged Heuristic Search (BS*)

a. General Description

This technique can use any of the basic methods listed in the first paragraph of this chapter. What makes this approach unique is how that basic method is used. BS* is actually two searches: one starting at the starting point and working towards the goal, the

other starting at the goal and working towards the starting point. After each iteration a "wave" of acceptable moves is generated for each search. When the two waves meet at a connecting point a path from the starting point to the goal is completed. Figure 2-3 illustrates the BS*. [Kwa 89]

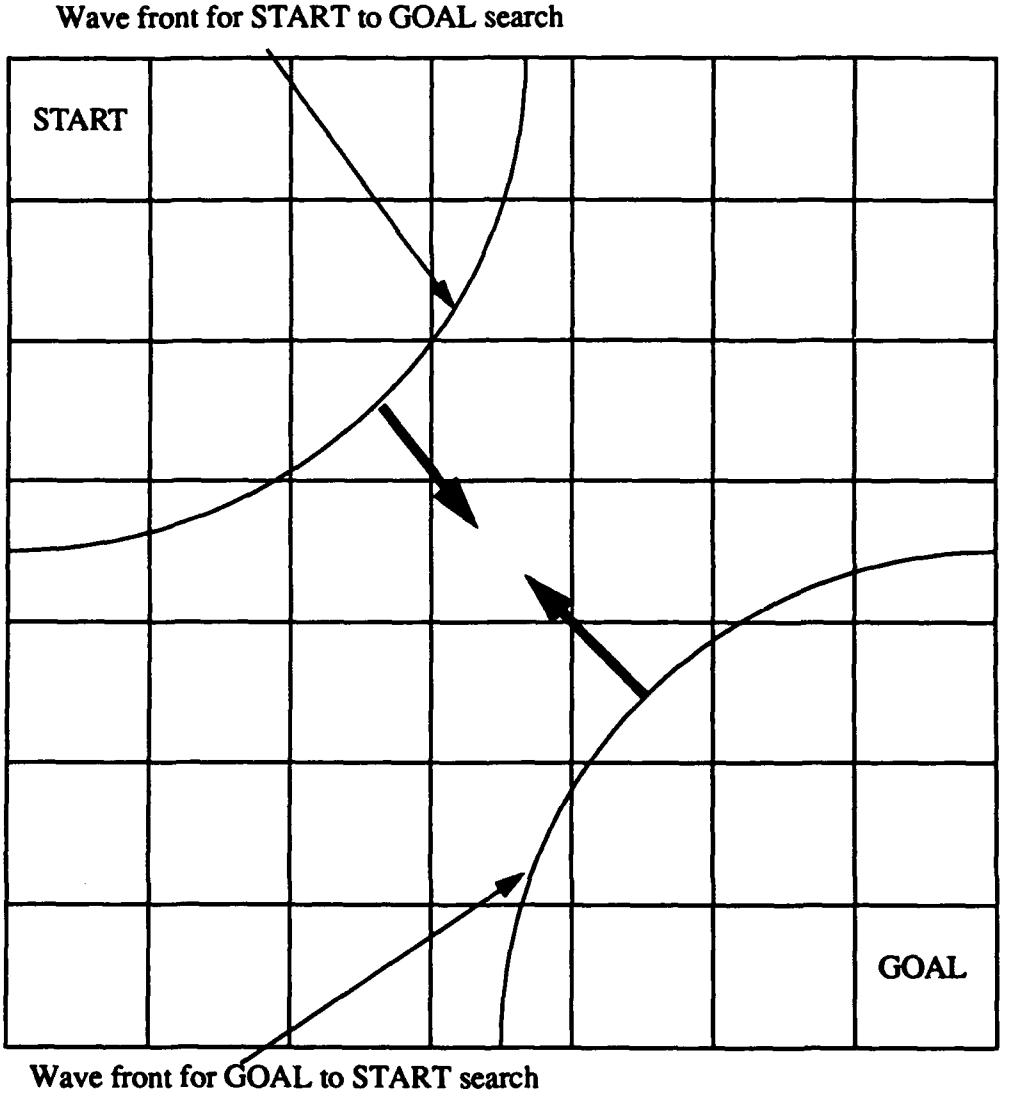


Figure 2-3 Bidirectional Search Process

b. Trees

As the search expands on a node finding its subsequent legal moves, a tree-like structure is generated. Here the branching factor becomes a problem: the trees grow exponentially. Various methods, like pruning and trimming have been used to reduce the size of the trees and provide more efficient processing. It has been proposed that generating two trees, vice one, reduces the total effort of processing the search space. Therefore, by dividing the search process into two halves (start-to-goal, and goal-to-start) the trees generated are overall smaller than that generated by a single direction search. [Kwa 89]

c. Cost

With the reduction of tree size and search space, it would be logical to assume the process to require less time to generate a path. This, however, is not the case. After each iteration a check must be made to determine if the two search halves have met. This check can be a costly process and may not reduce overall computational time.

Other researchers have tried to "push" or "nudge" the search tree growth along an expected path and thereby reduce the number of "open" nodes. This, however, can lead to a non-admissable solution to the problem and a less then optimal path may be generated. Kwa uses nipping, pruning, trimming and screening to help reduce the number of open nodes and reduce run-time. Nipping, pruning, trimming and screening are techniques used to eliminate paths that are obviously too costly.

d. Advantages and Disadvantages

The most noteworthy advantage is that this method can be executed on a parallel processing computer. With a multi-processor system such as the T-800 transputer, each processor could perform a search. Taking this one step further, consider a path with an intermediate way-point as indicated in Figure 2-4. A multi-processor system could process each path segment (start-to-waypoint, waypoint-to-start, goal-to-waypoint, waypoint-to-goal) on different processors. The same problem of checking for the connecting point is still unavoidable.

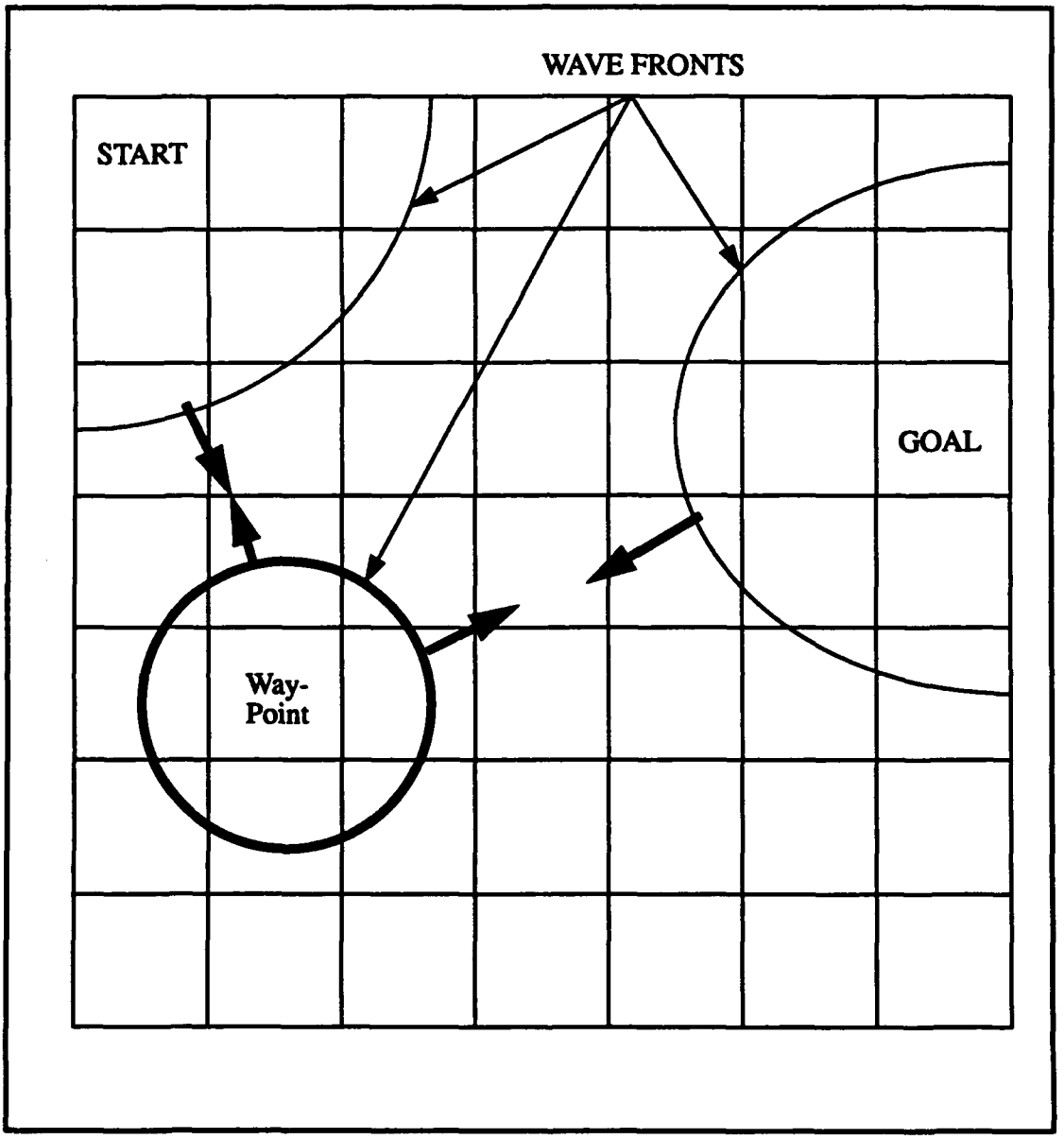


Figure 2-4 Bidirectional Search Process with Waypoint

4. Configuration Space (C-space)

a. General Description

The underlying assumption of C-space path planning is that it is easier to plan for a point size vehicle then for a rigid body vehicle. By reducing the vehicle representation to a point, the world model must be altered so that a safe path can be planned. How the world is changed is the interesting aspect of this technique. [Warren 90]

b. C-Space Obstacles

Since the rigid body vehicle is represented as a point something must be done to insure a safe path is planned [Lozano-Perez 83]. To accomplish this, obstacle size is altered to reflect a vehicle size buffer as shown in Figure 2-5 [Latombe 91]. It is important to note that obstacle buffer size varies depending on the vehicle orientation. Each orientation specific obstacle is called a shield and can be calculated at run-time to reduce preprocessing workload. [Latombe 91]

c. Conclusion

C-space obstacle representation may be good where very precise navigation is necessary. Calculations for vehicle representation are reduced by representing the vehicle as a point but computational time is increased by the requirement for calculating the shields for each orientation. The four-dimensional Tendril search uses a very simplified C-space concept. Orientation is limited to the four cardinal headings and each node is represented by four shields.

5. Potential Field

a. General Description

Given a girded or nodal search space, potentials are assigned to each node. These potentials are based upon proximity of a node to the goal or an obstacle. Nodes with obstacles in close proximity will have a repulsion potential since it is undesirable to position a vehicle in these nodes. Other nodes provide a free and clear path to the goal and

thus are assigned an attraction potential. Once the potential field is established any search technique can be used to find the least cost path.[Warren 90]

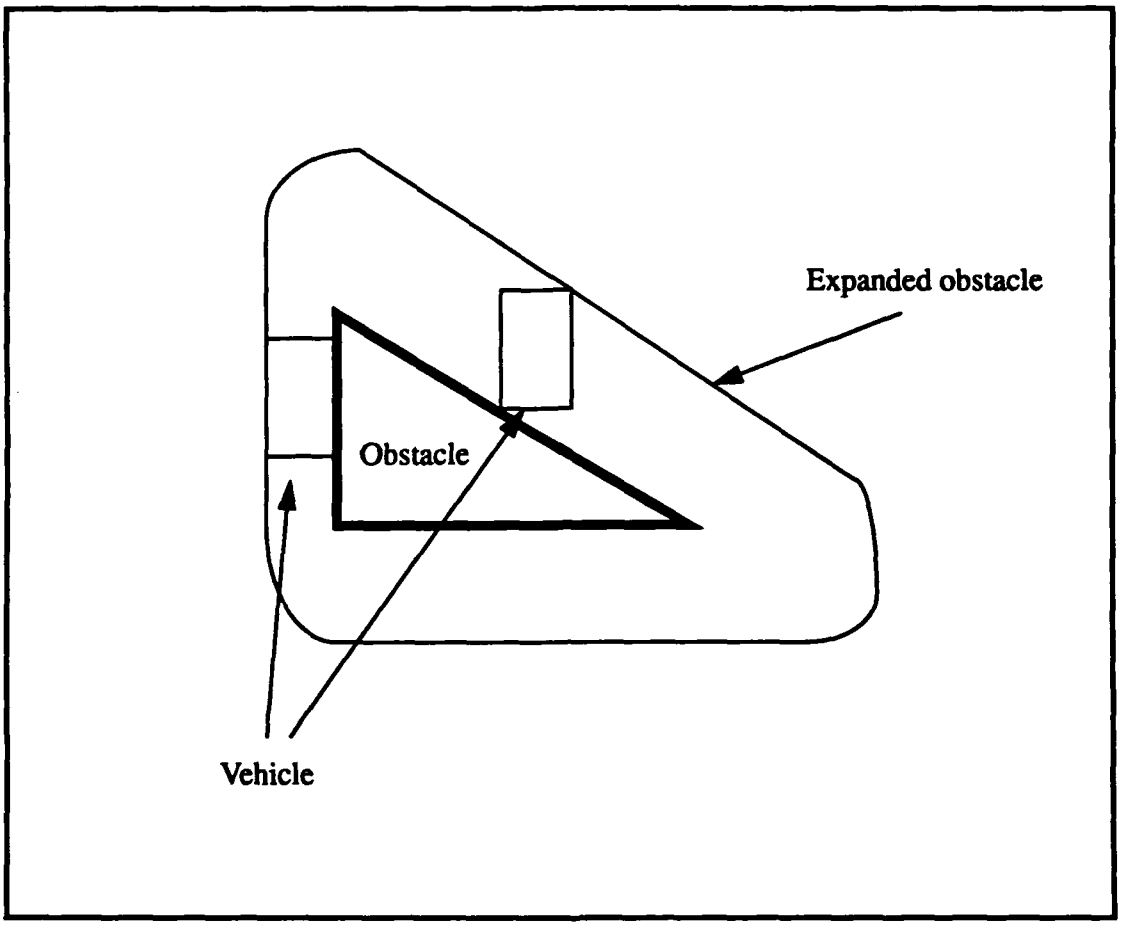


Figure 2-5 Expanded C-Space Obstacle for a Vehicle with Fixed Orientation

b. World Representation

Figure 2-6 shows how a potential field can be represented. Free space is assigned a potential based on the square of the distance from the goal. Nodes near obstacles are assigned a potential based on the reciprocal of the distance from the obstacle, squared. After preprocessing the search space, assigning potentials, a topology or landscape is developed that allows a vehicle to travel "downhill" to the goal.[Warren 90]

c. Conclusion

Again, local maxima can cause a problem by "trapping" the search. Figure 2-6 displays this and indicates the need for an alternate search method. In the figure, the shortest path would be along the diagonal between the start and goal. The potentials, however may drive the path in a less optimal direction or worse; trap the path at the local maxima. The Direction search incorporates the potential field concept at a very abstract level. Values are not assigned to each node, however, a "pointer" to the next node in the shortest path to the goal is stored at each node in the search space.

6. Remarks

Each of the techniques presented have been researched well and the use of multiple methods has been explored, yet combination of techniques in a single search is somewhat rare. Each has its merit and could easily be incorporated in varying degrees of complexity to produce an efficient path planner. The search techniques to be presented in the following chapters use some variation of these techniques. The predominant difference from previous research is that some of these techniques are combined into a single search process.

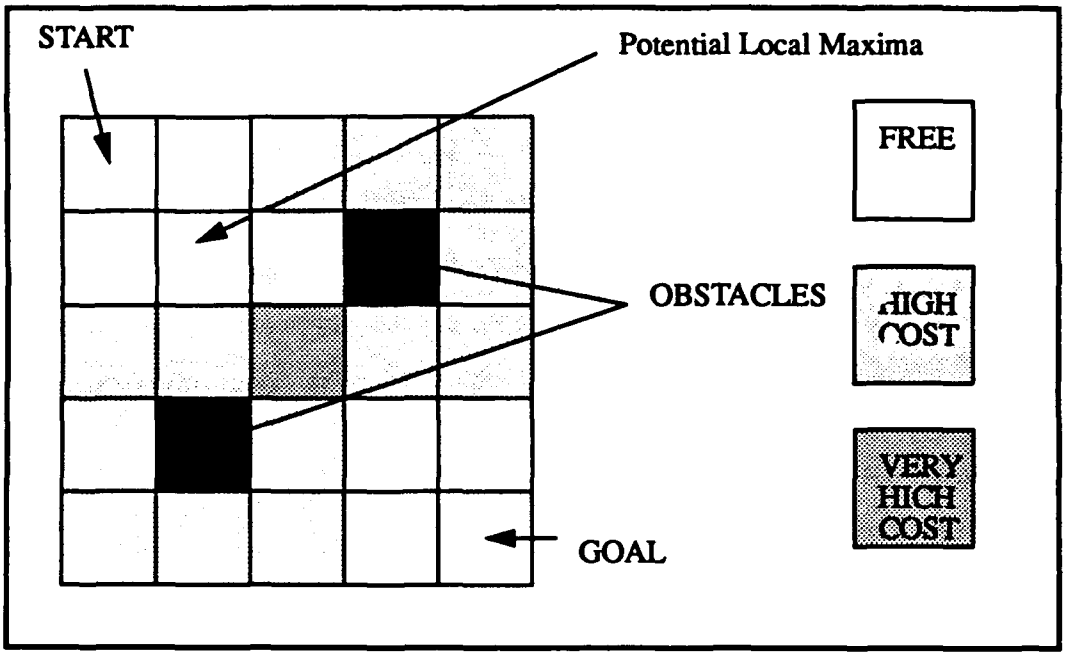


Figure 2-6 Example Potential Field Representation

III. THE TENDRIL SEARCH

A. GENERAL

In the Tendril search, the search space is represented by an multi-dimensional array, or lattice of nodes. Each node maintains several attributes to facilitate the path planning process. One of the attributes is a list of the node's immediate neighbors, with an associated cost to move to each neighbor. Only non-obstacle neighbors are maintained in this list and represent the legal moves that can be made from the node. The legal moves from the starting point represent the first WAVE. Legal moves are found for each node of this WAVE and the process continues generating subsequent waves until the goal is reached.

B. LISP VERSION

1. Two Dimensional Problem

Originally written for a two dimensional search space [McGhee 90], there are eight potential legal moves (one for each cardinal direction and one for each diagonal move) that can be made from the starting point. These legal moves are placed in an "open" list, WAVE, of nodes to be expanded upon. The next wave is determined by finding the legal moves for each node in the WAVE list. As each node is processed it is assigned a tendril length representing the length of the current path from the start to that specific node. There is an exponential increase in the number of nodes for each wave which could cause limitations due to memory requirements. Since individual nodes can be reached via multiple paths, it is important to consider each one so the shortest can be selected. By pruning previously processed nodes whose assigned tendril length is less than that of the current waves calculation for the tendril length, the longer and redundant paths are eliminated. This specifies that the node can be reached by a shorter path and the current path need not be investigated. The pruning process helps to reduce the time required to

preprocess the search space. Even so, preprocessing of the search space for each node's legal successor is not feasible, especially for real-time constraints.

2. Four Dimensional Problem

When a third dimension (depth) is considered, the problem becomes very complex. Each node has 26 possible legal moves. This is unacceptable if real-time path planning is needed. To reduce this number an additional dimension is considered, heading. It is not unreasonable and is only natural to consider orientation when planning a path. Only the cardinal headings (North, East, West and South) are considered in this program. Since a vehicle must have a heading the number of legal moves can be reduced to nine as shown in Figure 3-1. Thus the two dimensional problem is easily expanded to four dimensions with little increase in computational complexity. More details of this program are found in [Bonsignore 90]. Appendix C lists the four Dimensional LISP code (3dh.lisp).

C. ADA VERSION DESCRIPTION

1. General

"Why use Ada?" is a question surely posed by some researchers. Through a recent mandate, the government requires the use of Ada for its software projects. This, however, is not the driving force behind the use of Ada for this thesis. Ada was designed for use in large programming projects. With characteristics such as separate compilation and generic procedures, it facilitates the modularization of programming projects and allows several programmers to work individually. Modularity also helps with program maintainability. Ada provides multitasking and timing constructs which facilitate real-time systems programming [Voltz, et al 84]. All of these attributes make Ada especially suitable for AUV programming.

The initial intent of this thesis was to build a path planner with Ada reusable code. Some difficulties with reusable code were encountered. The Ada software repository at White Sands Missile Range is not quite "user friendly". Very general procedures, such as building linked lists, were acceptable. More sophisticated code, however, was often

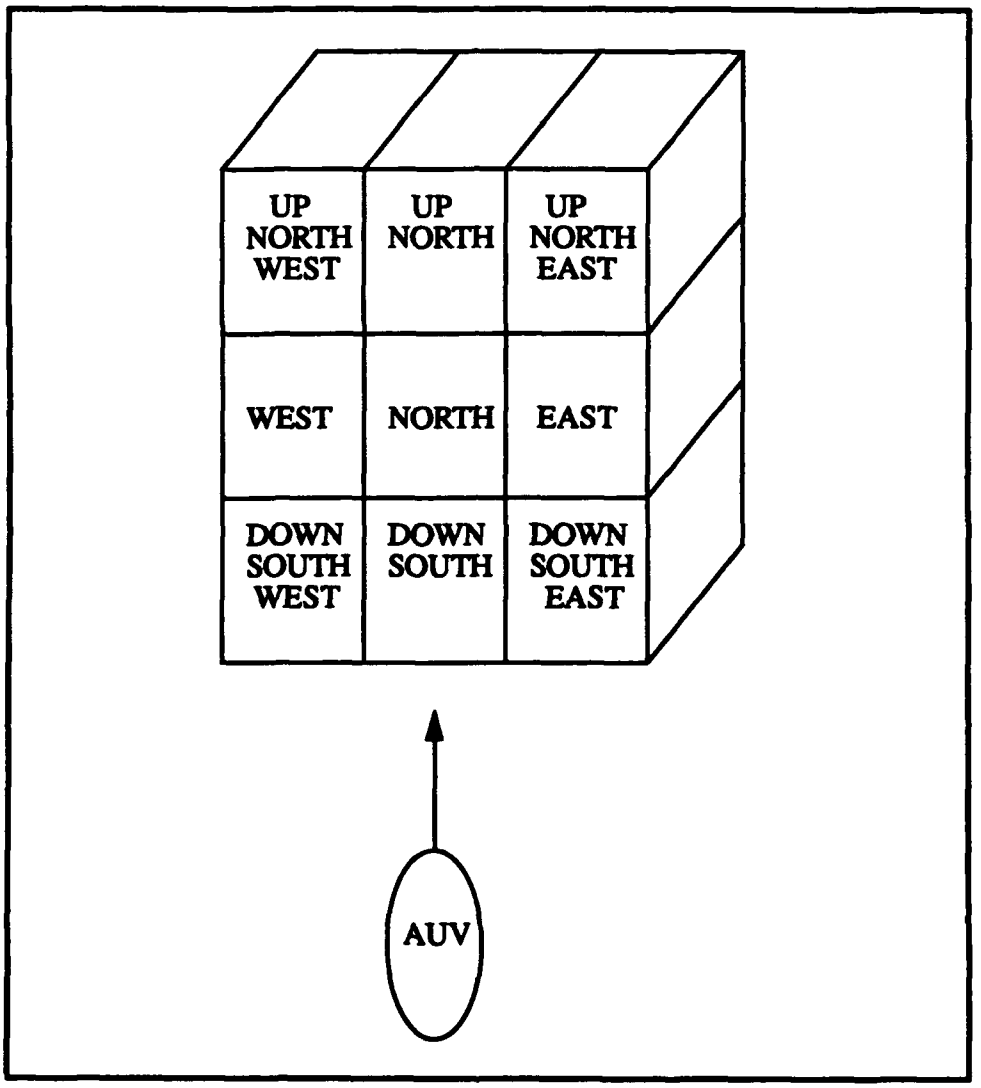


Figure 3-1 Nine Legal Moves When Heading is Considered (Heading North)

difficult to find. Some of these procedures were not stored with "user friendly" file names and often stored with coded names, making quick access difficult. Even when the appropriate code was obtained and the module found to be usable, many modifications were required. On large projects these modifications and subsequent testing may be more costly then writing the procedures from scratch [Gaffney 89]. For these reasons this thesis does not take advantage of reusable Ada software.

2. Direct Translation

An initial attempt at programming in Ada was made by making a direct translation from LISP. All the LISP structures and functions were translated into Ada records and procedures. This task was not easy since the two languages are very different. Many modifications, although small, were required in the Ada code due to these language differences.

a. Memory Problem

A major difficulty with the direct translation was that of memory usage. The LISP version preprocesses the search space assigning a list of possible legal moves from each node. In a two dimensional problem there are eight legal moves for each node (the test search space is a 10 x 20 array resulting in approximately 1600 legal moves). The memory requirement for this can be too large for some systems (as was the case for a modestly configured 386SX). Preprocessing is wasteful since nodes that do not require processing were processed anyway. These problems were solved by modification in the Ada version which will be presented in subsequent sections of this chapter.

b. Speed

Due to the extensive search process requirements, the speed at which the LISP version ran was slow. The directly translated Ada version did not run at all due to system memory constraints, therefore no timing characteristics are available.

3. Four Dimensional Problem

Expanding the two dimensional problem to incorporate depth and heading drastically increased memory requirements since the number of legal moves over tripled resulting in a combinatorial explosion. As earlier stated, this was solved by considering only those moves that the vehicle can immediately transition to when the vehicles heading is taken into account. By considering orientation in the search process unnecessary path searches were eliminated.

4. Modifications

Many modifications were required to enable the four dimensional Ada Tendril search to run. Some changes were very simple and others required a complete rewrite to achieve the efficiency required. Appendix D is a data dictionary, DFD, and code for this process.

a. Smaller records

The LISP version record structure for a node maintained a list of all legal moves possible from that node. By eliminating this attribute the size of the record was substantially reduced thus easing the limitations imposed by memory restrictions. Section *b.* below describes how the legal moves are determined.

b. "F_MOVES"

To reduce memory requirements the preprocessing of the search space was eliminated. Instead, the legal moves were determined as each node was reached in the search process. Thus if looking for a path between two adjacent nodes the legal moves for the nodes far removed from the possible path are not determined. Listed below, is the pseudo-code representation of this process.

```
procedure F_MOVES (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR) is
```

```
  HEADING  : INT_TYPE := ROOT.LOC(4);

begin
 case HEADING is
  when the heading is north=>
   check the upper northwest node
   check the upper north node
   check the upper northeast node
   check the northwest node
   check the north node
   check the northeast node
   check the lower northwest node
   check the lower north node
   check the lower northeast node
  when heading east => ...
  when heading south =>...
  when heading west =>...
  when others =>
   null;
 end case;
end F_MOVES;

procedure F_PATH (N_ARRAY : in out NODE_ARRAY) is

 ROOT        : LIST_PTR := WAVE;

begin
 while the root contains valid information loop
  F_MOVES (N_ARRAY, ROOT);
  ROOT := WAVE.NEXT;
 end loop;
 if the goal is found then
  return to the main process (DO_SEARCH)
 end if;
end F_PATH;
```

F_MOVES, called from within F_PATH, takes as input the search space (N_ARRAY) and the current node being processed (ROOT). Using the ROOT's heading the nine legal moves are determined. Each legal move is processed and if the GOAL is among them, the search is complete, otherwise they are assigned to a list called WAVE. The legal moves for each

node in WAVE are then generated and the process of checking for GOAL is repeated. This continues until the goal is found or all nodes are processed without reaching the GOAL.

c. Waypoint capability

It is conceivable that an AUV may need to navigate to some intermediate points that may not be along the optimal path between the start and goal. For this reason a waypoint capability is required. The algorithm below illustrates the TENDRILWP search in pseudo-code which allows multiple waypoint path generation.

```
procedure TENDRILWP is

begin
  GET_DATA;
  DO_SEARCH;
end TENDRILWP

procedure GET_DATA is

begin
while there are still points to enter loop
 get way point coordinates
 exit when done
end loop

procedure DO_SEARCH is

begin
 read in the terrain data
create the output file
loop
   exit when the second node in WAVE is null
   while WAVE is not empty and the goal isn't found
    F_PATH
 end loop;
 print the path
 reset the search space and variables to initials values
end loop
 close the output file
end DO_SEARCH
```

The tendril search technique is used in each path planning segment. A "while...loop" construct was added to the DO_SEARCH procedure (in the PATHWP package). Each waypoint is processed in order, finding a path between each of two successive points. The generated path segment is printed to a file and the next two points are processed until complete. Upon completion of a path segment many variables need to be reset to their initial values to allow the next path segment to be generated. The RESET_ALL procedure resets global variables.

This path planning technique divides the search into several small searches lending itself to concurrent processing. While one processor finds a path from the start to the waypoint, another processor could find the path from the waypoint to the goal. This version does not take advantage of concurrent processing, it finds each path segment sequentially. Appendix E contains the Data Dictionary, DFD and program code.
Limitations

As previously indicated, memory and speed have continued to be a concern in this methods efficiency [Richbourg et. al. 87]. If the whole search space needs to reside in memory, it is restricted by the machines capabilities. If time constraints permit, reading and writing to a file may be a feasible solution. This aspect was not investigated in this thesis. It is interesting to note that as the obstacle density increases, a larger search space can be stored in memory without causing memory problems. This is a result of "pruning" the obstacle nodes from the legal move and open node lists. As the number of obstacle nodes increases the free space is logically decreased.

5. The Tendril Algorithm

The Tendril search takes as input the starting coordinates including orientation (row, col, dep, hdg) and the goal. Terrain data is read from a file and is implemented in a dynamic array described by the first few items (array dimensions) read from the file. From the starting point and consistent with the initial heading, the legal moves are determined and assigned to WAVE. Each node in WAVE is assigned a parent node (in this case the

parent is the starting point) and its tendril length is calculated from the tendril length of the parent plus the distance from the parent to that node. Checks are made to ensure nodes previously processed are not reprocessed unless the resulting tendril length is shorter than the nodes current tendril length. This process is performed on each node in WAVE which generates another wave. Iteratively, it continues until a wave reaches the goal. Once the wave containing the goal is fully processed, the program stops the search and backtracks from the goal, via its parent "pointer" to the start, printing out the path (or writing it to a file). Listed below is the pseudo-code of the Tendril search.

```
procedure TENDRIL is

begin
  GET_DATA;
  DO_SEARCH;
end TENDRIL;

procedure GET_DATA is

begin
  get the terrain file name
  get array data
  get START and GOAL coordinates
end GET_DATA

procedure DO_SEARCH is

begin
  get the terrain data
  while the WAVE list is not empty loop
    F_PATH
    exit when the goal is found
  end loop;
  print the path
end DO_SEARCH
```

As previously described the F_MOVES procedure performs most of the work. It checks for the ROOT's heading and uses a case statement to handle each of the cardinal headings. For example, when the vehicle is heading north (HDG = 1) only the nine nodes

in a northerly direction are evaluated. These evaluations are performed in the THE_MOVES package. Each CHECK_ *NODE* (where *NODE* represents one of the nine legal moves) procedure determines the coordinates of the legal node being evaluated and calls the CK_STATE procedure. In CK_STATE the node's state is determined to be either free space or obstacle space. If the node is free space, the GROW_TEND process is called. This process adjusts the tendril length, assigns the parent (ROOT), and attaches that node to the NEW_WAVE list. Other procedures are used to process the WAVE and NEW_WAVE lists in support of the F_MOVE procedure.

D. TENDRIL BIDIRECTIONAL SEARCH

1. Concept

Although not implemented in this research the Tendril Bidirectional search (TBS) appears to be a viable solution to real-time processing problems. This method lends itself well to concurrent processing as previously described for the Tendril search with waypoints. With the installation of a T-800 transputer board into the NPS AUV II, concurrent processing is highly desirable and achievable.

2. Limitations

An important consideration is the need to check for completion after each wave iteration. This could be a difficult problem, reducing the advantages of concurrency by requiring a high degree of communication between each search process.

E. EVALUATION AND RESULTS

For ease of comparison, a smaller terrain representation was used in the evaluation (5 rows X 5 columns X 5 depths X 4 headings). When tested with an obstacle free model, the paths generated in both the Ada and LISP (compiled) versions were identical. The Ada code was much faster (.24 seconds vice .933 seconds for LISP). It is especially significant considering the following facts: While the LISP version has the terrain data hard coded, the Ada version must open and read the data from a disk file. The Ada version writes its results

to the screen and to a file. Both of these differences are I/O processes which are time intensive. Even with these I/O hindrances the Ada version was the fastest. In an obstacle intense terrain model, results were similar.

IV. VECTOR FIELD METHOD

A. GENERAL DESCRIPTION

Although a very simple process, this method has proven to be the most efficient of those investigated in this thesis. Listed below, is the pseudo-code for the DIRECTION procedure.

```
procedure DIRECTION is

begin
 GET_DATA
 DO_DIR
end DIRECTION

procedure GET_DATA is

begin
 get the terrain file name
 get array data
 get START and GOAL coordinates
end GET_DATA

procedure DO_DIR is

begin
 get the terrain data
 FIND_MOVES
 FIND_PATH
 P_PATH
end DO_DIR
```

The legal moves are determined by searching backwards from the goal. The node attribute, NEXT, stores the coordinates of a successor node having the shortest distance to traverse. As legal moves are determined the NEXT attribute is assigned the coordinates of the node being expanded. Therefore the nodes generated from the goal will have the goals coordinates stored in the NEXT attribute. Figure 4-1 illustrates the backwards search process and direction assignment for a three dimensional problem (row, column, and

heading) with the GOAL having a southerly orientation. The numbers in each node represent the order that they were processed during the search. Two nodes adjacent to the GOAL remain unassigned because it is impossible for these nodes to move directly to the

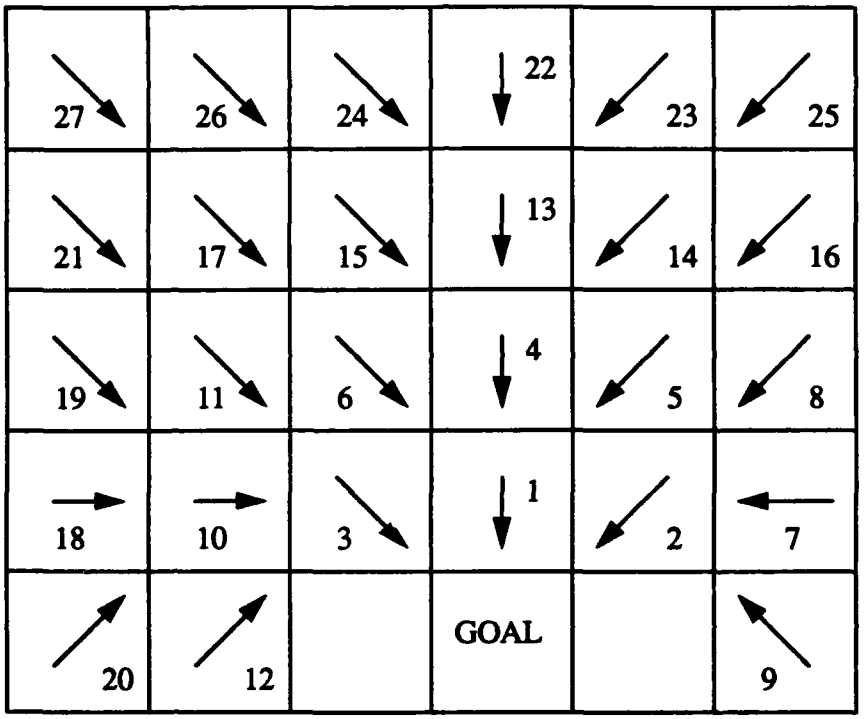


Figure 4-1 Representation of a Vector Field

GOAL with the proper orientation. This is not the case in the four dimensional problem (including depth). A transition in depth will allow all paths to the GOAL to be generated.

Nodes are processed with a priority. A move not requiring a heading change is less expensive than moves that do. Nodes are put into a search queue based on the cost to move into the next node. The entire search space is processed this way, resulting in "vectors" being assigned to every free space node. Once the starting point is entered, all moves to the goal are immediately available.

B. ADVANTAGES AND DISADVANTAGES

The most obvious advantage is the speed at which this process runs as compared to the Tendril search. To find a path, the user provides the starting point. Each individual node "knows" its next move to reach the goal. Path determination is just a matter of following the NEXT attributes until the goal is reached. No calculations are required during the path planning process making it very fast.

Other searches need both the starting point and the goal to preprocess the search space. The Direction process needs only the goal to prepare the search space. Since the preprocessing is not dependent on the starting point, many trials with various starting points can be investigated with only the cost of the preprocessing once.

Another advantage is the ease at which obstacles are handled during the process. Originally written for only non-obstacle terrain, a small modification was required to handle obstacles (i.e. an obstacle is an illegal NEXT move).

As can be seen in Figure 4-1, all the paths to the GOAL may not be generated. If the starting point is the node just to the right of the GOAL, a path may not be found (where one exists) to reach the GOAL with the appropriate orientation (southerly heading). This disadvantage can be overcome when considering a four dimensional problem.

C. BASIC PROGRAM FLOW

Similar to the Tendril search, terrain, starting point and goal information are taken as input. Some of the procedures are exactly the same as those used in the Tendril search, while others required minor modifications. Most notably is that FIND_MOVES procedure processes legal moves in "reverse" from the F_MOVES procedure in the Tendril search. A pseudo-code version of this procedure is listed below. It finds all the legal moves from which the goal can be reached as opposed to which node can be move into from the starting point. The legal moves are placed into a queue of active nodes, ACTIVE, based upon a predefined order relative to heading. The ordering results in the nodes with the least cost being at the head of the queue and the rest follow in increasing cost order. Each member of

the ACTIVE queue is processed in a similar manner with its legal moves being appended to the end of the queue. As each node is processed its NEXT attribute is assigned the coordinates of its parent. This is, essentially, a pointer to the shortest move to attain the

```
procedure FIND_MOVES (N_ARRAY : in out NODE_ARRAY) is

 heading  : INT_TYPE := ACTIVE.LOC(4);
 list     : LOC_ARRAY := ACTIVE.LOC;
 new_list : LOC_ARRAY := ACTIVE.LOC;

begin
 while the ACTIVE list is not empty loop
  heading := ACTIVE.LOC(4);
  list := ACTIVE.LOC;
  new_list := ACTIVE.LOC;
  case heading is
   when heading north =>
    check the southern node
    check the upper south node
    check the lower south node
    check the southeast node
    check the upper southeast node
    check the lower southeast node
    check the southwest node
    check the upper southwest node
    check the lower southwest node
   when heading east => ...
   when heading south => ...
   when heading west => ...
   when others =>
    null
  end case;
end FIND_MOVES;
```

goal. Processing the entire search space results in every free space node being assigned a NEXT node to move to and a cost associated with that move. Obstacles are not processed and a NEXT move cannot be assigned the coordinates of an obstacle. The A_AND_A (analyze and assign) procedure insures the assignment of the NEXT attribute is done

properly. The P_PATH process generates the path. Beginning with the starting node, it follows the NEXT "pointers" until the goal is reached and writes the nodes to a file. Appendix E contains the Data Dictionary, DFD, and program code.

D. THE DIRECTION SEARCH ALGORITHM

Producing most of the work for this technique is the, previously mentioned, FIND_MOVES procedure. Having passed in N_ARRAY (the array of nodes) it uses the GOAL and a case statement to determine legal moves. It is very similar to the F_MOVES procedure in the Tendril search except that it works in "reverse." Looking at the orientation required in the GOAL, it determines what node an AUV can transition from to attain that GOAL. Procedures in the THE_MOVES package then determine the coordinates of these nodes and calls the A_AND_A procedure (analyze and assign). In this procedure it is determined if the nodes are free or obstacle space. If a node is free space it is assigned a value, DIST, equal to the distance that must be traversed to enter that node. It is also assigned to the ACTIVE queue in a specified order as previous detailed. Other procedures, similar to the Tendril search, are supporting means for processing the ACTIVE list.

E. RESULTS AND EVALUATION

Similar to the timing results of the Ada version of the Tendril search, the Direction search is significantly faster then the LISP Tendril search. The results of a search conducted in an obstacle free space produced similar results, although slightly faster (.2271 seconds vice .233 for Ada Tendril). This timing variance may be explained by I/O differences. A significant timing difference was noted between the searches in an obstacle intense environment. The Direction search was much faster (.11 seconds vice .329 seconds). This is attributed to the different way obstacles are handled in the two programs. The Direction search has much less overhead for handling obstacles.

V. PATH REPLANNING

A. GENERAL DESCRIPTION

A path replanner is a path planner with more stringent time constraints. It is needed when an AUV is required to circumnavigate an unexpected obstacle to continue its mission. This replanner must, therefore, operate in real-time to facilitate an efficient transition to an alternate path.

The Real-time A* (RTA*) algorithm presented by Korf [Korf 88] was modified to incorporate four dimensions. This method can use any of the previously mentioned techniques for searching (Best-first, Tendril, etc.) but only searches to a specified search depth. Nodes at the search depth are called frontier nodes.The method implemented in this thesis uses the Tendril search to a search depth of three nodes. As the search progresses, the cost of reaching the frontier nodes is calculated. Adding this cost to an estimate to reach the goal, the frontier node with the lowest overall cost is chosen to be expanded upon. The process is repeated at this intermediate frontier node and successively until the goal is reached. Appendix F contains the Data Dictionary, DFD and program code. Pseudo-code for the RTA* search is listed below.

```
procedure RTA is

begin
 GET_DATA
 DO_SEARCH
end RTA

procedure DO_SEARCH is

begin
 get the terrain data from file
 while the goal is not found loop
  find the frontier nodes
```

```
  pick the node with the estimated least cost
 end loop
 print the path
end DO_SEARCH
```

B. JUSTIFICATION

Previously presented search methods may not be efficient enough for real-time path replanning. As a possible solution to this problem, the RTA* technique was investigated. Due to the nature of the AUV's working environment it is feasible to "expect the unexpected." The dynamic nature of the undersea environment can alter terrain and obstacles swiftly, rendering preprogrammed paths obsolete. For this reason an on-board replanner is required which must operate in real-time.

Many considerations went into the implementation of the RTA* for this thesis:

1. What search method should be used to find the frontier nodes?
2. What should the search depth be?
3. Would the old path be completely disregarded or should a new path try to return to the old path as soon as possible?
4. Should this procedure handle the initial collision avoidance maneuver?
5. How "real" is real-time?

These questions had to be properly answered to produce a true real-time path replanner. Since this thesis predominately examined the Tendril search, it was determined that it should be used for the search method in the RTA*. The search depth was arbitrarily chosen at five nodes and later reduced to three because of memory limitations. Old path data is discarded and the new path does not attempt to "get back on" the old path. Initial collision avoidance is to be performed by a different procedure and this RTA* would be a path replanner only. Strict real-time constraints have not been set.

C. DISADVANTAGES

Although good for two dimensional problems, the RTA* is very memory intensive for multi-dimensional problems. As waves are processed the legal moves for all nodes up to the search depth must be retained. To illustrate the exponential growth of search nodes, consider a starting point that has only three legal moves in its first frontier. At a two node frontier distance (or second frontier) 27 legal moves must be maintained and 243 legal moves at a three frontier. A combinatorial explosion results with further processing. For a moving vehicle, a three node length look ahead capability may be insufficient (depending on node size) for obstacle avoidance reaction time.

D. EVALUATION

Due to the massive memory requirements, this process appeared to be fruitless especially when the good results of the previously presented methods are considered. Upon a second consideration, it is feasible to incorporate pruning methods to help eliminate unnecessary processing of nodes which should not be processed. Pruning techniques were successfully used in the other methods and therefore should not be difficult to implement into the RTA* search. Further research with this method is recommended.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY AND CONCLUSIONS

This thesis investigated several path planning techniques using a nodal representation of the search space. A four dimensional Tendril search was implemented in Ada and a time comparison to a LISP version was made. The results indicate that it is feasible to use Ada for intelligent, real-time path planning. One version of the Tendril search incorporated a waypoint capability. It is an important aspect that should be looked at more carefully for implementation in the NPS AUV II.

The Direction search, using a vector field was implemented. This method proved to be the fastest of the methods investigated. Due to its speed and simplicity, it is highly recommended for the NPS AUV II replanner for the near term.

The RTA* search initially appeared to be cumbersome for multi-dimensional path replanning. Upon reconsideration, it could be modified to take advantage of pruning techniques to eliminate unnecessary node processing.

B. RECOMMENDATIONS

Each method of path planning investigated was valuable for various reasons. Simplicity was the prevailing aspect in all methods which, in turn, resulted in small time requirements for search space processing. Of these procedures the Direction search was the simplest and fastest, thus recommended for further research to incorporate into the NPS AUV II as the onboard path replanner.

It should be noted that there are limited orientation capabilities for each method. Although only the cardinal headings were used, the results are sufficient for path-planning purposes. The guidance module of the NPS AUV II does not use the orientations produced in the path planning process. It uses only three dimensional coordinates, generating

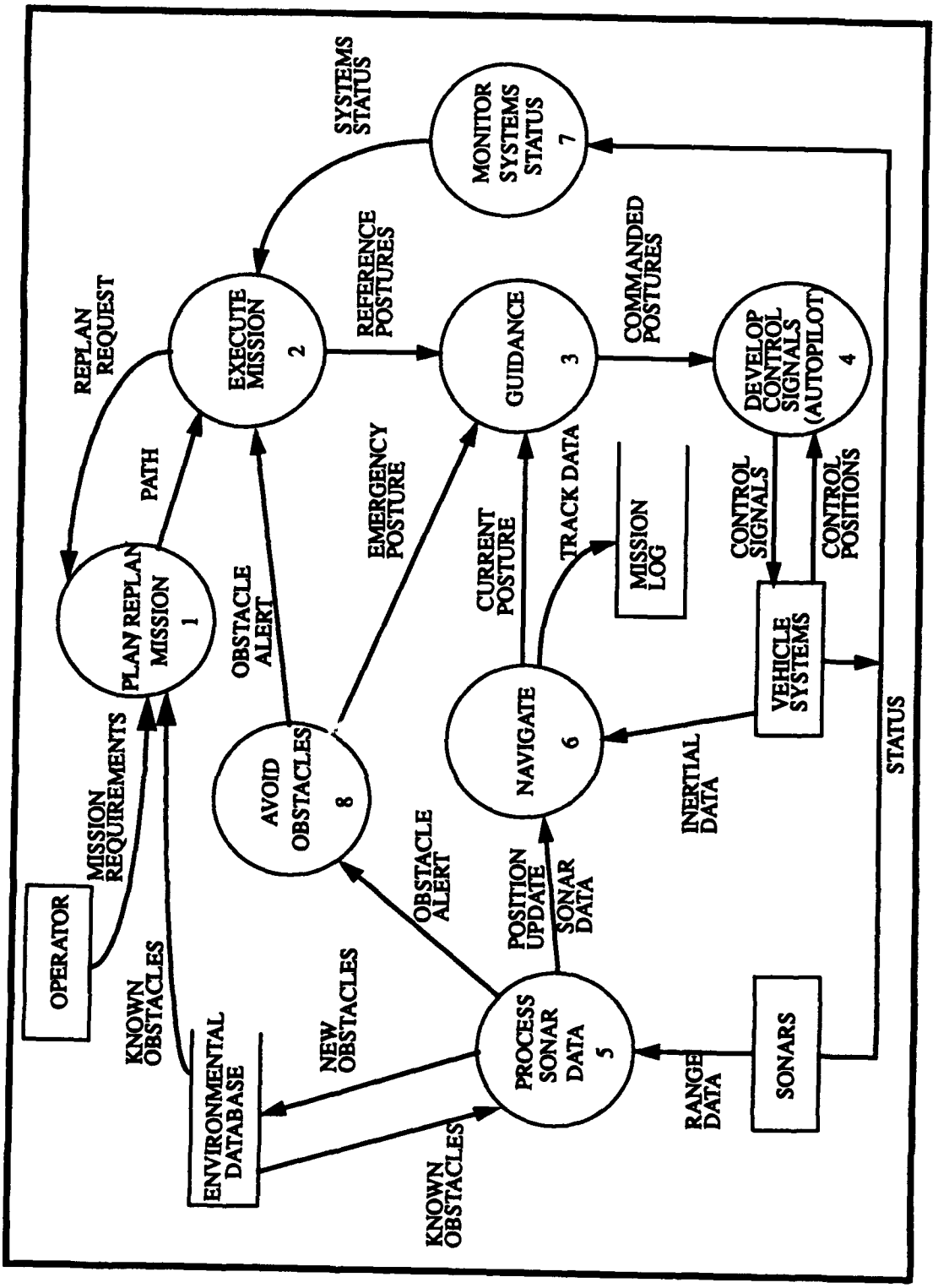
orientation more accurately itself. Thus, the course orientations of these path planners are only for their internal use to accommodate more accurate planning. [Magrino 91]

The use of waypoints is a very important feature of path planning, whether for an aircraft, AUV, or a trip to the corner market. Use of the Direction search with a waypoint capability is recommended for further research. It may be difficult or time restrictive since the vector field generated depends on the goal. A new vector field is required for each path segment. Consideration should be given to dividing the search space into portions, each path segment having its own portion of the search space eliminating the requirement to reinitialize the entire search space after each path segment is planned.

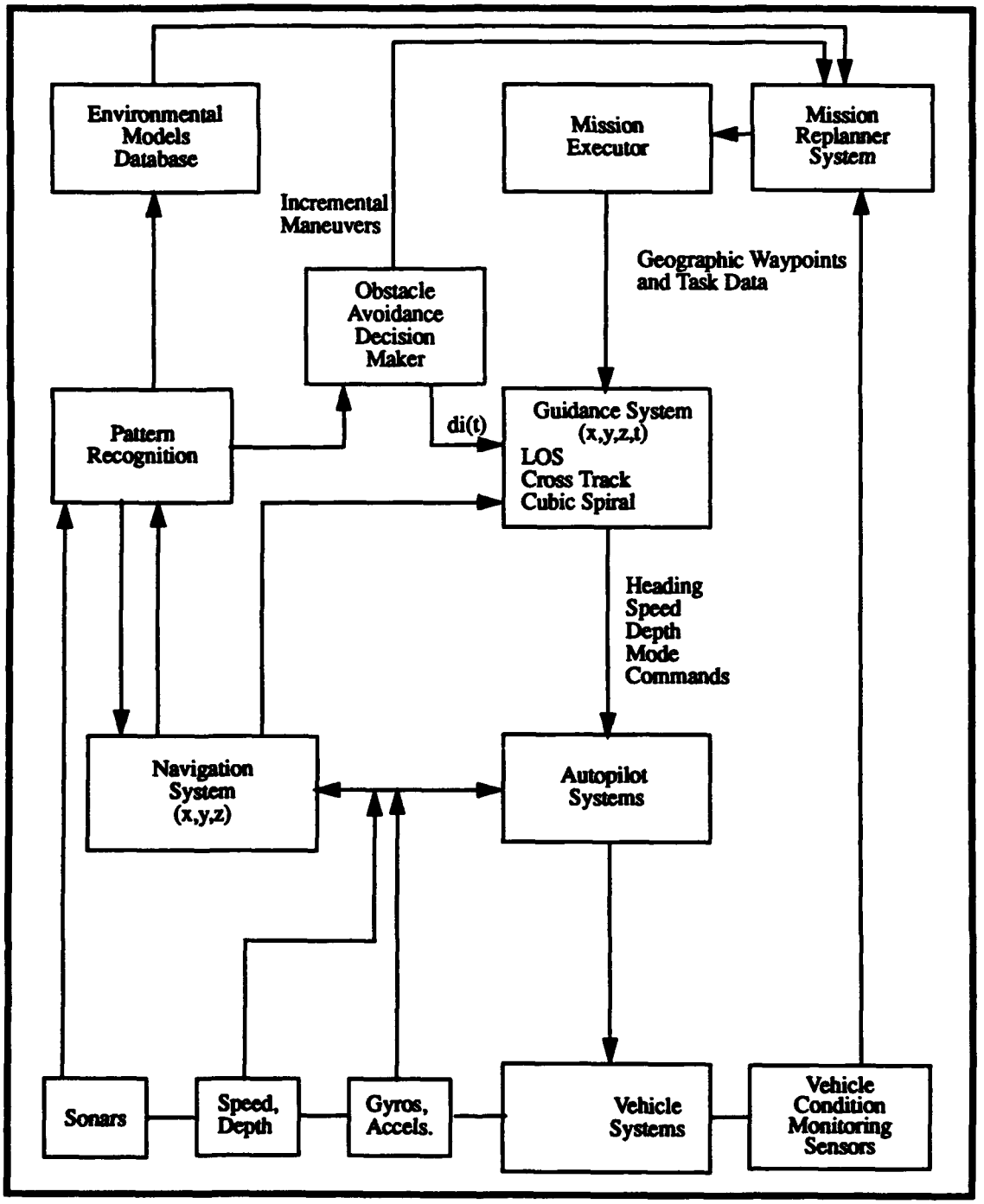
The Tendril Bidirectional Search (TBS) lends itself to concurrent processing. The use of transputers could make this an exceptional method for real-time path planning.

Consideration should be given to the use of multiple path planning and replanning methods. Planning for obstacle intense environments is significantly different from obstacle sparse environments. In most cases it appeared that the easiest path to plan (no obstacles) took the longest time. The Mission Planning Expert System has the capability to determine appropriate planning methods yet it has very few methods implemented. Further research is required to build upon the MPES path planning methods.

APPENDIX A - Data Flow Diagram for the NPS AUV II



APPENDIX B -
NPS AUV II System Block Diagram



APPENDIX C
Three Dimensional Tendril Search in LISP

```
defstruct node state parent tendril-length link-list)
(defvar *new-active-node-list* nil)
(defvar *active-node-list* nil)
(defvar *goal* nil)
(defvar *goal-flag* nil)
(defvar *node-array* (make-array '(6 7 7 7)))
(defvar *cycle-number* 0)
(defvar *terrain* (make-array '(6 7 7 7) :initial-contents (ommitted)

(defun create-node (h k i j)
 (setf (aref *node-array* h k i j) (make-node)))

(defun initialize-state (h k i j)
 (if (= 1 (aref *terrain* h k i j))
 (setf (node-state (aref *node-array* h k i j)) 'obstacle)))

(defun set-state (heading depth row column state)
 (setf (node-state (aref *node-array* heading depth row column)) state))

(defun set-parent (heading depth row column parent)
 (setf (node-parent (aref *node-array* heading depth row column)) parent))

(defun set-tendril-length (heading depth row column length)
 (setf (node-tendril-length (aref *node-array* heading depth row column)) length))

(defun set-link-list (heading depth row column list)
 (setf (node-link-list (aref *node-array* heading depth row column)) list))

(defun state (heading depth row column)
 (node-state (aref *node-array* heading depth row column)))

(defun parent (heading depth row column)
 (node-parent (aref *node-array* heading depth row column)))

(defun tendril-length (heading depth row column)
 (node-tendril-length (aref *node-array* heading depth row column)))

(defun link-list (heading depth row column)
 (node-link-list (aref *node-array* heading depth row column)))
```

```
(defun make-terrain (heading-size depth-size row-size column-size)
 (dotimes (h heading-size 'terrain-initialized)
 (dotimes (k depth-size)
 (dotimes (i row-size)
 (dotimes (j column-size)
 (create-node h k i j)
 (initialize-state h k i j))))))

(defun legal-fwd-connected-link-list (heading depth row column)
 (non-nil-cons (4-link heading depth (1- row) column)
 (non-nil-cons (diag-link-ul 2 depth (1- row) (1- column))
 (non-nil-cons (diag-link-ur 4 depth (1- row) (1+ column))
 (non-nil-cons (diag-up-link-ul 2 (1- depth) (1- row) (1- column))
 (non-nil-cons (diag-up-link-u heading (1- depth) (1- row) column)
 (non-nil-cons (diag-up-link-ur 4 (1- depth) (1- row) (1+ column))
 (non-nil-cons (diag-down-link-ul 2 (1+ depth) (1- row) (1- column))
 (non-nil-cons (diag-down-link-l heading (1+ depth) (1- row) column)
 (non-nil-cons (diag-down-link-ur 4 (1+ depth) (1- row) (1+ column)) nil))))))))))

(defun legal-left-connected-link-list (heading depth row column)
 (non-nil-cons (4-link heading depth row (1- column))
 (non-nil-cons (diag-link-ul 1 depth (1- row) (1- column))
 (non-nil-cons (diag-link-ll 3 depth (1+ row) (1- column))
 (non-nil-cons (diag-up-link-ul 1 (1- depth) (1- row) (1- column))
 (non-nil-cons (diag-up-link-l heading (1- depth) row (1- column))
 (non-nil-cons (diag-up-link-ll 3 (1- depth) (1+ row) (1- column))
 (non-nil-cons (diag-down-link-ul 1 (1+ depth) (1- row) (1- column))
 (non-nil-cons (diag-down-link-l heading (1+ depth) row (1- column))
 (non-nil-cons (diag-down-link-ll 3 (1+ depth) (1+ row) (1- column)) nil))))))))))

(defun legal-back-connected-link-list (heading depth row column)
 (non-nil-cons (4-link heading depth (1+ row) column)
 (non-nil-cons (diag-link-ll 2 depth (1+ row) (1- column))
 (non-nil-cons (diag-link-lr 4 depth (1+ row) (1+ column))
 (non-nil-cons (diag-up-link-ll 2 (1- depth) (1+ row) (1- column))
 (non-nil-cons (diag-up-link-d heading (1- depth) (1+ row) column)
 (non-nil-cons (diag-up-link-lr 4 (1- depth) (1+ row) (1+ column))
 (non-nil-cons (diag-down-link-ll 2 (1+ depth) (1+ row) (1- column))
 (non-nil-cons (diag-down-link-d heading (1+ depth) (1+ row) column)
 (non-nil-cons (diag-down-link-lr 4 (1+ depth) (1+ row) (1+ column)) nil))))))))))

(defun legal-right-connected-link-list (heading depth row column)
 (non-nil-cons (4-link heading depth row (1+ column))
```

```
(non-nil-cons (diag-link-lr 3 depth (1+ row) (1+ column))
(non-nil-cons (diag-link-ur 1 depth (1- row) (1+ column))
(non-nil-cons (diag-up-link-lr 3 (1- depth) (1+ row) (1+ column))
(non-nil-cons (diag-up-link-r heading (1- depth) row (1+ column))
(non-nil-cons (diag-up-link-ur 1 (1- depth) (1- row) (1+ column))
(non-nil-cons (diag-down-link-lr 3 (1+ depth) (1+ row) (1+ column))
(non-nil-cons (diag-down-link-r heading (1+ depth) row (1+ column))
(non-nil-cons (diag-down-link-ur 1 (1+ depth) (1- row) (1+ column)) nil))))))))))

(defun 4-link (heading depth row column)
 (if (not (equal (state heading depth row column) 'obstacle))
 (list (list heading depth row column) 2)))

(defun diag-link-ul (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading depth row (1+ column)) 'obstacle))
 (not (equal (state heading depth (1+ row) column) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-link-ll (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading depth (1- row) column) 'obstacle))
 (not (equal (state heading depth row (1+ column)) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-link-lr (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading depth (1- row) column) 'obstacle))
 (not (equal (state heading depth row (1- column)) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-link-ur (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading depth row (1- column)) 'obstacle))
 (not (equal (state heading depth (1+ row) column) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-up-link-ul (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading (1+ depth) row column) 'obstacle))
 (not (equal (state heading depth (1+ row) (1+ column)) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))
```

```
(defun diag-up-link-l (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading (1+ depth) row column) 'obstacle))
 (not (equal (state heading depth row (1+ column)) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-up-link-ll (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading (1+ depth) row column) 'obstacle))
 (not (equal (state heading depth (1- row) (1+ column)) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-up-link-d (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading (1+ depth) row column) 'obstacle))
 (not (equal (state heading depth (1- row) column) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-up-link-lr (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading (1+ depth) row column) 'obstacle))
 (not (equal (state heading depth (1- row) (1- column)) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-up-link-r (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading (1+ depth) row column) 'obstacle))
 (not (equal (state heading depth row (1- column)) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-up-link-ur (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading (1+ depth) row column) 'obstacle))
 (not (equal (state heading depth (1+ row) (1- column)) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-up-link-u (heading depth row column)
 (if (and (not (equal (state heading depth row column) 'obstacle))
 (or (not (equal (state heading (1+ depth) row column) 'obstacle))
 (not (equal (state heading depth (1+ row) column) 'obstacle))))
 (list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-down-link-ul (heading depth row column)
```

```
(if (and (not (equal (state heading depth row column) 'obstacle))
(or (not (equal (state heading (1- depth) row column) 'obstacle))
(not (equal (state heading depth (1+ row) (1+ column)) 'obstacle))))
(list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-down-link-l (heading depth row column)
(if (and (not (equal (state heading depth row column) 'obstacle))
(or (not (equal (state heading (1- depth) row column) 'obstacle))
(not (equal (state heading depth row (1+ column)) 'obstacle))))
(list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-down-link-ll (heading depth row column)
(if (and (not (equal (state heading depth row column) 'obstacle))
(or (not (equal (state heading (1- depth) row column) 'obstacle))
(not (equal (state heading depth (1- row) (1+ column)) 'obstacle))))
(list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-down-link-d (heading depth row column)
(if (and (not (equal (state heading depth row column) 'obstacle))
(or (not (equal (state heading (1- depth) row column) 'obstacle))
(not (equal (state heading depth (1- row) column) 'obstacle))))
(list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-down-link-lr (heading depth row column)
(if (and (not (equal (state heading depth row column) 'obstacle))
(or (not (equal (state heading (1- depth) row column) 'obstacle))
(not (equal (state heading depth (1- row) (1- column)) 'obstacle))))
(list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-down-link-r (heading depth row column)
(if (and (not (equal (state heading depth row column) 'obstacle))
(or (not (equal (state heading (1- depth) row column) 'obstacle))
(not (equal (state heading depth row (1- column)) 'obstacle))))
(list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-down-link-ur (heading depth row column)
(if (and (not (equal (state heading depth row column) 'obstacle))
(or (not (equal (state heading (1- depth) row column) 'obstacle))
(not (equal (state heading depth (1+ row) (1- column)) 'obstacle))))
(list (list heading depth row column) (* 2 (sqrt 2)))))

(defun diag-down-link-u (heading depth row column)
(if (and (not (equal (state heading depth row column) 'obstacle))
```

```
(or (not (equal (state heading (1- depth) row column) 'obstacle))
(not (equal (state heading depth (1+ row) column) 'obstacle))))
(list (list heading depth row column) (* 2 (sqrt 2)))))

(defun non-nil-cons (item list)
(if (null item) list (cons item list)))

(defun initialize-heading-connected-map (heading-size depth-size row-size column-size)
(make-terrain heading-size depth-size row-size column-size)
(dotimes (h (- heading-size 2))
(dotimes (k (- heading-size 2))
(dotimes (i (- row-size 2))
(dotimes (j (- column-size 2))
(set-tendril-length (1+ h) (1+ k) (1+ i) (1+ j) 0)
(if (= (1+ h) 1)
(set-link-list (1+ h) (1+ k) (1+ i) (1+ j)
(legal-fwd-connected-link-list (1+ h) (1+ k) (1+ i) (1+ j))))
(if (= (1+ h) 2)
(set-link-list (1+ h) (1+ k) (1+ i) (1+ j)
(legal-left-connected-link-list (1+ h) (1+ k) (1+ i) (1+ j))))
(if (= (1+ h) 3)
(set-link-list (1+ h) (1+ k) (1+ i) (1+ j)
(legal-back-connected-link-list (1+ h) (1+ k) (1+ i) (1+ j))))
(if (= (1+ h) 4)
(set-link-list (1+ h) (1+ k) (1+ i) (1+ j)
(legal-right-connected-link-list (1+ h) (1+ k) (1+ i) (1+ j)))))))))

(defun update-root-node (heading depth row column new-tendril-length new-link-list)
(set-tendril-length heading depth row column new-tendril-length)
(set-link-list heading depth row column new-link-list))

(defun activate-end-node (root node residue)
(if (equal node *goal*) (setf *goal-flag* t))
(set-state (first node) (second node) (third node) (fourth node) *cycle-number*)
(set-parent (first node) (second node) (third node) (fourth node) root)
(set-tendril-length (first node) (second node) (third node) (fourth node) residue)
(setf *new-active-node-list* (cons node *new-active-node-list*)))

(defun verify-parent (root node residue)
(when (> residue (tendril-length (first node) (second node) (third node) (fourth node)))
(set-parent (first node) (second node) (third node) (fourth node) root)
(set-tendril-length (first node) (second node) (third node) (fourth node) residue)))
```

```
(defun test-link (root link tendril-length)
 (let ((residue (- tendril-length (second link)))
(end-node-state (state (first (first link)) (second (first link)) (third (first link)) (fourth (first
link)))))
 (cond ((and (null end-node-state) (>= residue 0))
 (activate-end-node root (first link) residue) nil)
 ((and (numberp end-node-state) (= *cycle-number* end-node-state)
 (>= residue 0))
 (verify-parent root (first link) residue) nil)
 ((null end-node-state) link))))

(defun grow-tendrils (root tendril-increment)
 (let* ((heading (first root)) (depth (second root)) (row (third root)) (column (fourth root))
 (new-tendril-length (+ tendril-increment (tendril-length heading depth row column)))
 (new-link-list nil))
 (dolist (link (link-list heading depth row column)
 (update-root-node heading depth row column new-tendril-length new-link-list))
 (setf new-link-list (non-nil-cons (test-link root link new-tendril-length)
  new-link-list)))))

(defun increment-wavefront (tendril-increment) ;returned value not used
 (dolist (root *active-node-list* *new-active-node-list*)
 (setf *new-active-node-list*
 (non-nil-cons (test-root root tendril-increment) *new-active-node-list*))))

(defun test-root (root tendril-increment) ;returns root if any tendrils alive
 (if (grow-tendrils root tendril-increment) root))

(defun find-path (start goal tendril-increment)
 (initialize-heading-connected-map 6 7 7 7)
 (set-state (first start) (second start) (third start) (fourth start) 0)
 (setf *goal* goal *cycle-number* 0 *active-node-list* (list start)
 *goal-flag* nil)
 (loop (if (or (null *active-node-list*) (not (null *goal-flag*)))
 (return (if (not (null *goal-flag*)) (pprint (path-to-goal goal)))))
 (setf *new-active-node-list* nil)
 (setf *cycle-number* (1+ *cycle-number*))
 (setf *active-node-list* (increment-wavefront tendril-increment))))

(defun path-to-goal (goal)
 (let ((parent (parent (first goal) (second goal) (third goal) (fourth goal))))
 (if parent (cons goal (path-to-goal parent)))))
```

APPENDIX D (part 1)

Table 1: Data Dictionary for the Tendril Search

PACKAGE PROCEDURE VARIABLE	TYPE
GLOBALS DATA_FILE PATH_FILE	 FILE_TYPE
LOC_ARRAY	array (1..4) of INT_TYPE
LIST	record {LOC : LOC_ARRAY INC : INT_TYPE NEXT : LIST_PTR}
NODE	record {STATE : INT_TYPE PARENT : LOC_ARRAY TEND_LEN : INT_TYPE}
NODE_ARRAY	array (INT_TYPE range <>, INT_TYPE range <>, INT_TYPE range <>, INT_TYPE range <>) of NODE
START_TIME END_TIME T_TIME	TIME DURATION
MAX_ROW MAX_COL MAX_DEP MAX_HDG DIAG_COST := 99 CARD_COST := 70	INT_TYPE

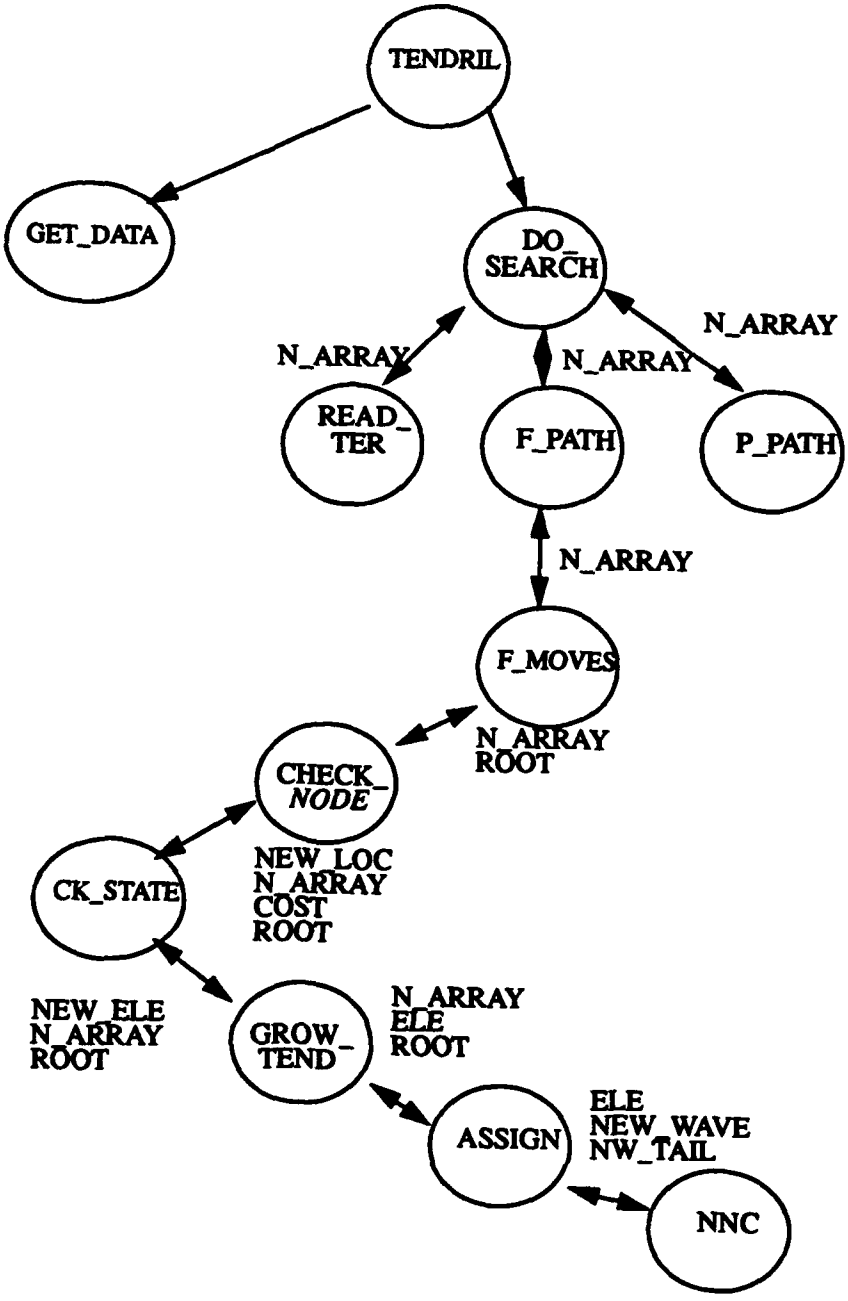
Table 1: Data Dictionary for the Tendril Search

PACKAGE PROCEDURE VARIABLE	TYPE
WAVE NEW_WAVE NW_TAIL LAST_NEW_WAVE WAVE_HEAD WAVE_TAIL THE_PATH THE_PATH_CURRENT TRASH	LIST_PTR := null
START GOAL	LOC_ARRAY
GOAL_FOUND	BOOLEAN := FALSE
PATH GET_DATA FILE_NAME	 STRING (1..12)
NAME_LEN	INT_TYPE
P_PATH NEXT_LOC	 LOC_ARRAY
F_MOVES HEADING	 INT_TYPE := ROOT.LOC (4)
F_PATH ROOT	 LIST_PTR := WAVE
DO_SEARCH N_ARRAY	 NODE_ARRAY (1..MAX_ROW, 1..MAX_COL, 1..MAX_DEP, 1..MAX_HDG)

Table 1: Data Dictionary for the Tendril Search

PACKAGE PROCEDURE VARIABLE	TYPE
THE_MOVE CK_STATE NEW_ELE	 LIST_PTR
CHECK_*NODE* NEW_LOC	 LOC_ARRAY := ROOT.LOC

DATA FLOW DIAGRAM for the TENDRIL SEARCH (part 2)



TENDRIL SEARCH CODE (part 3)

```
--=========================================================
--NAME        :  J. Bonsignore, Jr.
--DATE        :  22 Jan, 1991
--REVISED     :
--TITLE       :  TENDRIL.ADA
--DESCRIPTION :  Main procedure for the Tendril search
--CALLS       :  GET_DATA and DO_SEARCH in the PATH package
--NOTES       :
--=========================================================

with TEXT_IO, GLOBALS, PATH;
use TEXT_IO, GLOBALS, PATH;

procedure TENDRIL is

begin
  GET_DATA;
  DO_SEARCH;
end TENDRIL;
```

```
--============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  GLOBALS.ADS
--DESCRIPTION  :  Global variables for the searches
--CALLS        :
--NOTES        :
--============================================================

with TEXT_IO, CALENDAR;
use TEXT_IO, CALENDAR;

package GLOBALS is

  subtype INT_TYPE is INTEGER;
  package INT_IO is new INTEGER_IO (INT_TYPE);
  package FLOATIO is new FLOAT_IO (FLOAT);
  use INT_IO, FLOATIO;

  type LOC_ARRAY is array (1..4) of INT_TYPE;

  type LIST;
  type LIST_PTR is access LIST;
  type LIST is
    record
           LOC  : LOC_ARRAY:= (others => 0);
           INC  : INT_TYPE := 0;
           NEXT : LIST_PTR;
    end record;

  type NODE;
  type NODE_PTR is access NODE;
  type NODE is
    record
           STATE    : INT_TYPE  := 0;
           PARENT   : LOC_ARRAY := (others => 0);
           TEND_LEN : INT_TYPE  := 0;
    end record;

 type NODE_ARRAY is array (INT_TYPE range <>, INT_TYPE range
                           <>,INT_TYPE range <>, INT_TYPE
                           range <>) of NODE;
```

```
   DATA_FILE   : FILE_TYPE;
   PATH_FILE   : FILE_TYPE;

   START_TIME  : TIME;
   END_TIME    : TIME;

   T_TIME      : DURATION;

   MAX_ROW     : INT_TYPE;
   MAX_COL     : INT_TYPE;
   MAX_DEP     : INT_TYPE;
   MAX_HDG     : INT_TYPE;

   DIAG_COST   : INT_TYPE := 99;
   CARD_COST   : INT_TYPE := 70;

   WAVE           : LIST_PTR := null;
   NEW_WAVE       : LIST_PTR := null;
   NW_TAIL        : LIST_PTR := null;
   LAST_NEW_WAVE  : LIST_PTR := null;

   WAVE_HEAD         : LIST_PTR := null;
   WAVE_TAIL         : LIST_PTR := null;
   THE_PATH          : LIST_PTR := null;
   THE_PATH_CURRENT  : LIST_PTR := null;

   TRASH : LIST_PTR := null;

   START       : LOC_ARRAY;
   GOAL        : LOC_ARRAY;

   GOAL_FOUND : BOOLEAN := FALSE;

end GLOBALS;
```

```
--=============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  PATH.ADS
--DESCRIPTION  :
--              :
--CALLS        :
--NOTES        :
--=============================================================

with TEXT_IO, GLOBALS, THE_MOVE,
UNCHECKED_DEALLOCATION,CALENDAR;
use TEXT_IO, GLOBALS, THE_MOVE, CALENDAR;

package PATH is

  procedure DO_SEARCH;

  procedure GET_DATA;

  procedure READ_TER (N_ARRAY : in out NODE_ARRAY);

  procedure P_PATH (N_ARRAY : in out NODE_ARRAY);

end PATH;

--=============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  PATH.ADB
--DESCRIPTION  :
--              :
--CALLS        :
--NOTES        :
--=============================================================

package body PATH is

    procedure GET_DATA is

      FILE_NAME : STRING (1..12);
```

```
  NAME_LEN  : INT_TYPE;

begin
  put ("Enter the name of data file:  ");
  get_line (FILE_NAME, NAME_LEN);
  FILE_NAME ((NAME_LEN + 1)..12) := (others => ' ');
  FILE_NAME (9..12) := ".DAT";
  OPEN (DATA_FILE, MODE => IN_FILE, NAME => FILE_NAME);
  INT_IO.get (DATA_FILE, MAX_ROW);
  INT_IO.get (DATA_FILE, MAX_COL);
  INT_IO.get (DATA_FILE, MAX_DEP);
  INT_IO.get (DATA_FILE, MAX_HDG);
  NEW_LINE;
  put ("Enter the starting row:  ");
  INT_IO.get (START(1));
  NEW_LINE;
  put ("Enter the starting col:  ");
  INT_IO.get (START(2));
  NEW_LINE;
  put ("Enter the starting dep:  ");
  INT_IO.get (START(3));
  NEW_LINE;
  put ("Enter the starting hdg:  ");
  INT_IO.get (START(4));
  NEW_LINE;
  put ("Enter the goal row:  ");
  INT_IO.get (GOAL(1));
  NEW_LINE;
  put ("Enter the goal col:  ");
  INT_IO.get (GOAL(2));
  NEW_LINE;
  put ("Enter the goal dep:  ");
  INT_IO.get (GOAL(3));
  NEW_LINE;
  put ("Enter the goal hdg:  ");
  INT_IO.get (GOAL(4));
  WAVE := new LIST;
  WAVE.LOC := START;
  WAVE.INC := 0;
end GET_DATA;

procedure READ_TER (N_ARRAY : in out NODE_ARRAY) is
```

```
begin
     for ROW in 1..MAX_ROW loop
       for COL in 1..MAX_COL loop
            for DEP in 1..MAX_DEP loop
              for HDG in 1..MAX_HDG loop
               INT_IO.get (DATA_FILE, N_ARRAY(ROW, COL,
                           DEP, HDG).STATE);
              N_ARRAY(ROW, COL, DEP, HDG).TEND_LEN := 0;
               N_ARRAY(ROW, COL, DEP, HDG).PARENT :=
                                              (0,0,0,0);
              end loop;
            end loop;
       end loop;
     end loop;
     close (DATA_FILE);
end READ_TER;

procedure P_PATH (N_ARRAY : in out NODE_ARRAY) is

  NEXT_LOC  : LOC_ARRAY;
  PATH_FILE : FILE_TYPE;

begin
     if GOAL_FOUND then
       CREATE (PATH_FILE, NAME => "path.file");
       put ('(');
       INT_IO.put (GOAL(1));
       INT_IO.put (GOAL(2));
       INT_IO.put (GOAL(3));
       INT_IO.put (GOAL(4));
       put (')');
       INT_IO.put (PATH_FILE, GOAL(1));
       INT_IO.put (PATH_FILE, GOAL(2));
       INT_IO.put (PATH_FILE, GOAL(3));
       INT_IO.put (PATH_FILE, GOAL(4));
       new_line;
       new_line (PATH_FILE);
       NEXT_LOC := N_ARRAY(GOAL(1), GOAL(2), GOAL(3),
                           GOAL(4)).PARENT;
       while NEXT_LOC /= START loop
            put ('(');
            INT_IO.put (NEXT_LOC(1));
            INT_IO.put (NEXT_LOC(2));
```

```
            INT_IO.put (NEXT_LOC(3));
            INT_IO.put (NEXT_LOC(4));
            put (')');
            INT_IO.put (PATH_FILE, NEXT_LOC(1));
            INT_IO.put (PATH_FILE, NEXT_LOC(2));
            INT_IO.put (PATH_FILE, NEXT_LOC(3));
            INT_IO.put (PATH_FILE, NEXT_LOC(4));
          NEXT_LOC := N_ARRAY(NEXT_LOC(1), NEXT_LOC(2),
                              NEXT_LOC(3),NEXT_LOC(4)).PARENT;
            NEW_LINE;
            new_line (PATH_FILE);
        end loop;
        put ('(');
        INT_IO.put (START(1));
        INT_IO.put (START(2));
        INT_IO.put (START(3));
        INT_IO.put (START(4));
        put (')');
        INT_IO.put (PATH_FILE, START(1));
        INT_IO.put (PATH_FILE, START(2));
        INT_IO.put (PATH_FILE, START(3));
        INT_IO.put (PATH_FILE, START(4));
        new_line;
        new_line (PATH_FILE);
        INT_IO.put (N_ARRAY (GOAL(1), GOAL(2), GOAL(3),
                             GOAL(4)).TEND_LEN);
        new_line;
        CLOSE (PATH_FILE);
      else
        put ("PATH NOT FOUND");
        new_line;
      end if;
end P_PATH;

procedure F_MOVES (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR) is

        HEADING   : INT_TYPE := ROOT.LOC(4);

begin
        case HEADING is
          when 1 =>
                CHECK_UP_NW (N_ARRAY, ROOT);
```

```
            CHECK_UP_N (N_ARRAY, ROOT);
            CHECK_UP_NE (N_ARRAY, ROOT);
            CHECK_NW (N_ARRAY, ROOT);
            CHECK_N (N_ARRAY, ROOT);
            CHECK_NE (N_ARRAY, ROOT);
            CHECK_DOWN_NW (N_ARRAY, ROOT);
            CHECK_DOWN_N (N_ARRAY, ROOT);
            CHECK_DOWN_NE (N_ARRAY, ROOT);

        when 2 =>
            CHECK_UP_NE (N_ARRAY, ROOT);
            CHECK_UP_E (N_ARRAY, ROOT);
            CHECK_UP_SE (N_ARRAY, ROOT);
            CHECK_NE (N_ARRAY, ROOT);
            CHECK_E (N_ARRAY, ROOT);
            CHECK_SE (N_ARRAY, ROOT);
            CHECK_DOWN_NE (N_ARRAY, ROOT);
            CHECK_DOWN_E (N_ARRAY, ROOT);
            CHECK_DOWN_SE (N_ARRAY, ROOT);

        when 3 =>
            CHECK_UP_SE (N_ARRAY, ROOT);
            CHECK_UP_S (N_ARRAY, ROOT);
            CHECK_UP_SW (N_ARRAY, ROOT);
            CHECK_SE (N_ARRAY, ROOT);
            CHECK_S (N_ARRAY, ROOT);
            CHECK_SW (N_ARRAY, ROOT);
            CHECK_DOWN_SE (N_ARRAY, ROOT);
            CHECK_DOWN_S (N_ARRAY, ROOT);
            CHECK_DOWN_SW (N_ARRAY, ROOT);

        when 4 =>
            CHECK_UP_SW (N_ARRAY, ROOT);
            CHECK_UP_W (N_ARRAY, ROOT);
            CHECK_UP_NW (N_ARRAY, ROOT);
            CHECK_SW (N_ARRAY, ROOT);
            CHECK_W (N_ARRAY, ROOT);
            CHECK_NW (N_ARRAY, ROOT);
            CHECK_DOWN_SW (N_ARRAY, ROOT);
            CHECK_DOWN_W (N_ARRAY, ROOT);
            CHECK_DOWN_NW (N_ARRAY, ROOT);

        when others =>
```

```
                    null;

            end case;
    end F_MOVES;

    procedure FREE is new UNCHECKED_DEALLOCATION (LIST,
LIST_PTR);


    procedure F_PATH (N_ARRAY : in out NODE_ARRAY) is

            ROOT : LIST_PTR := WAVE;

    begin
            while ROOT /= null loop
              F_MOVES (N_ARRAY, ROOT);
              ROOT := WAVE.NEXT;
              FREE (WAVE);
              WAVE := ROOT;
            end loop;
            if GOAL_FOUND then
              return;
            end if;
            WAVE := NEW_WAVE;
            NEW_WAVE := null;
            LAST_NEW_WAVE := null;
    end F_PATH;

    procedure DO_SEARCH is

            N_ARRAY : NODE_ARRAY (1..MAX_ROW, 1..MAX_COL,
                                  1..MAX_DEP, 1..MAX_HDG);

    begin
         START_TIME := CLOCK;
           READ_TER (N_ARRAY);
           while WAVE /= null loop
             F_PATH (N_ARRAY);
             exit when GOAL_FOUND;
           end loop;
           P_PATH (N_ARRAY);
         END_TIME := CLOCK;
         T_TIME := END_TIME - START_TIME;
```

```
            NEW_LINE;
            FLOATIO.put (FLOAT(T_TIME));
            put ("  seconds of cpu time.");
        end DO_SEARCH;

    end PATH;
```

```
--=============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  THE_MOVE.ADS
--DESCRIPTION  :  Contains the procedures for checking and
--             :  processing individual
--             :  nodes sent from F_MOVES.
--CALLS        :
--NOTES        :
--=============================================================
with TEXT_IO, GLOBALS;
use TEXT_IO, GLOBALS;

package THE_MOVE is

  procedure NNC (ELEMENT : in LIST_PTR;
                 HEAD    : in out LIST_PTR;
                 TAIL    : in out LIST_PTR);

  procedure GROW_TEND (ELE     : in out LIST_PTR;
                       N_ARRAY : in out NODE_ARRAY;
                       ROOT    : in out LIST_PTR);

  procedure CK_STATE (NEW_LOC  : in out LOC_ARRAY;
                      N_ARRAY  : in out NODE_ARRAY;
                      NEW_INC  : in out INT_TYPE;
                      ROOT     : in out LIST_PTR);

  procedure CHECK_N (N_ARRAY : in out NODE_ARRAY;
                     ROOT    : in out LIST_PTR);

  procedure CHECK_UP_N (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR);

  procedure CHECK_DOWN_N  (N_ARRAY : in out NODE_ARRAY;
                           ROOT    : in out LIST_PTR);

  procedure CHECK_NE (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR);

  procedure CHECK_UP_NE (N_ARRAY : in out NODE_ARRAY;
                         ROOT    : in out LIST_PTR);
```

```
procedure CHECK_DOWN_NE (N_ARRAY : in out NODE_ARRAY;
                         ROOT    : in out LIST_PTR);

procedure CHECK_E (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR);

procedure CHECK_UP_E (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR);

procedure CHECK_DOWN_E (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR);

procedure CHECK_SE (N_ARRAY : in out NODE_ARRAY;
                    ROOT    : in out LIST_PTR);

procedure CHECK_UP_SE (N_ARRAY : in out NODE_ARRAY;
                       ROOT    : in out LIST_PTR);

procedure CHECK_DOWN_SE (N_ARRAY : in out NODE_ARRAY;
                         ROOT    : in out LIST_PTR);

procedure CHECK_S (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR);

procedure CHECK_UP_S (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR);

procedure CHECK_DOWN_S (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR);

procedure CHECK_SW (N_ARRAY : in out NODE_ARRAY;
                    ROOT    : in out LIST_PTR);

procedure CHECK_UP_SW (N_ARRAY : in out NODE_ARRAY;
                       ROOT    : in out LIST_PTR);

procedure CHECK_DOWN_SW (N_ARRAY : in out NODE_ARRAY;
                         ROOT    : in out LIST_PTR);

procedure CHECK_W (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR);
```

```
  procedure CHECK_UP_W (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR);

  procedure CHECK_DOWN_W (N_ARRAY : in out NODE_ARRAY;
                          ROOT    : in out LIST_PTR);

  procedure CHECK_NW (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR);

  procedure CHECK_UP_NW (N_ARRAY : in out NODE_ARRAY;
                         ROOT    : in out LIST_PTR);

  procedure CHECK_DOWN_NW (N_ARRAY : in out NODE_ARRAY;
                           ROOT    : in out LIST_PTR);

end THE_MOVE;

--=============================================================
--NAME        :  J. Bonsignore, Jr.
--DATE        :  22 Jan, 1991
--REVISED     :
--TITLE       :  THE_MOVE.ADB
--DESCRIPTION :  The package body for THE_MOVE
--=============================================================

with TEXT_IO, GLOBALS;
use TEXT_IO, GLOBALS;

package body THE_MOVE is

    procedure NNC (ELEMENT  : in LIST_PTR;
                   HEAD     : in out LIST_PTR;
                   TAIL     : in out LIST_PTR) is

--  Creates and performs list maintanence.
          begin
            if HEAD = null then
                   HEAD := ELEMENT;
                   TAIL := ELEMENT;
            else
                   TAIL.NEXT := ELEMENT;
                   TAIL := TAIL.NEXT;
            end if;
```

```
        end NNC;

    procedure GROW_TEND (ELE      : in out LIST_PTR;
                         N_ARRAY : in out NODE_ARRAY;
                         ROOT     : in out LIST_PTR) is

--  Determines if nodes have been previously processed and if
--  necessary reassignes.

        procedure ASSIGN (N_ARRAY : in out NODE_ARRAY;
                          ELE      : in out LIST_PTR;
                          ROOT     : in out LIST_PTR) is

--      Once a node is determined to be a legal move its
--      attributes
--      are assigned, and the GOAL is checked for completion.

           begin
             N_ARRAY(ELE.LOC(1),ELE.LOC(2),ELE.LOC(3),
                     ELE.LOC(4)).PARENT := ROOT.LOC;
             N_ARRAY(ELE.LOC(1),ELE.LOC(2),ELE.LOC(3),
                     ELE.LOC(4)).TEND_LEN := ELE.INC +
             N_ARRAY(ROOT.LOC(1),ROOT.LOC(2),ROOT.LOC(3),
                     ROOT.LOC(4)).TEND_LEN;
              if ELE.LOC = GOAL then
                     GOAL_FOUND := TRUE;
              end if;
              NNC (ELE, NEW_WAVE, NW_TAIL);
           end ASSIGN;

    begin
           if N_ARRAY(ELE.LOC(1),ELE.LOC(2),ELE.LOC(3),
           ELE.LOC(4)).TEND_LEN = 0 then
             ASSIGN (N_ARRAY, ELE, ROOT);
           elsif N_ARRAY(ELE.LOC(1),ELE.LOC(2),ELE.LOC(3),
                         ELE.LOC(4)).TEND_LEN >
           (N_ARRAY(ROOT.LOC(1),ROOT.LOC(2),ROOT.LOC(3),
                    ROOT.LOC(4)).TEND_LEN + ELE.INC) then
             ASSIGN (N_ARRAY, ELE, ROOT);
           end if;
    end GROW_TEND;

    procedure CK_STATE (NEW_LOC    : in out LOC_ARRAY;
```

```
                        N_ARRAY    : in out NODE_ARRAY;
                        NEW_INC    : in out INT_TYPE;
                        ROOT       : in out LIST_PTR) is

--  Checks if the node is an obstacle.  If not it calls the
--  GROW_TEND procedure.

        NEW_ELE : LIST_PTR;

    begin
        if N_ARRAY(NEW_LOC(1), NEW_LOC(2), NEW_LOC(3),
                   NEW_LOC(4)).STATE = 0 then
          NEW_ELE := new LIST;
          NEW_ELE.LOC := NEW_LOC;
          NEW_ELE.INC := NEW_INC;
          GROW_TEND (NEW_ELE, N_ARRAY, ROOT);
        end if;
    end CK_STATE;

--  The remaining procedures are for individual "moves."  The
--  coordinates are calculated and a cost is assigned to the
--  move.  Each calls CK_STATE.

    procedure CHECK_N (N_ARRAY : in out NODE_ARRAY;
                       ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

    begin
        IF NEW_LOC(1) > 1 then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          CK_STATE (NEW_LOC, N_ARRAY, CARD_COST, ROOT);
        end if;
    end CHECK_N;

    procedure CHECK_UP_N (N_ARRAY : in out NODE_ARRAY;
                          ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

    begin
        IF NEW_LOC(1) > 1 and NEW_LOC(3) > 1 then
          NEW_LOC(1) := NEW_LOC(1) - 1;
```

```
          NEW_LOC(3) := NEW_LOC(3) - 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_N;

procedure CHECK_DOWN_N (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) > 1 and NEW_LOC(3) < MAX_DEP then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_N;

procedure CHECK_NE (N_ARRAY  : in out NODE_ARRAY;
                    ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        if NEW_LOC(1) > 1 and NEW_LOC(2) < MAX_COL then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          NEW_LOC(2) := NEW_LOC(2) + 1;
          if ROOT.LOC(4) = 1 then
                NEW_LOC(4) := 2;
          else
                NEW_LOC(4) := 1;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end  CHECK_NE;

procedure CHECK_UP_NE (N_ARRAY : in out NODE_ARRAY;
                       ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) > 1 and NEW_LOC(2) < MAX_COL and
```

```
        NEW_LOC(3) > 1 then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          NEW_LOC(2) := NEW_LOC(2) + 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          if ROOT.LOC(4) = 1 then
                NEW_LOC(4) := 2;
          else
                NEW_LOC(4) := 1;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_NE;

procedure CHECK_DOWN_NE (N_ARRAY  : in out NODE_ARRAY;
                         ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) > 1 and NEW_LOC(2) < MAX_COL and
        NEW_LOC(3) < MAX_DEP then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          NEW_LOC(2) := NEW_LOC(2) + 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          if ROOT.LOC(4) = 1 then
                NEW_LOC(4) := 2;
          else
                NEW_LOC(4) := 1;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_NE;

procedure CHECK_E (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(2) < MAX_COL then
          NEW_LOC(2) := NEW_LOC(2) + 1;
          CK_STATE (NEW_LOC, N_ARRAY, CARD_COST, ROOT);
        end if;
```

```
end CHECK_E;

procedure CHECK_UP_E (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR) is

       NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
       IF NEW_LOC(2) < MAX_COL and NEW_LOC(3) > 1 then
         NEW_LOC(2) := NEW_LOC(2) + 1;
         NEW_LOC(3) := NEW_LOC(3) - 1;
         CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
       end if;
end CHECK_UP_E;

procedure CHECK_DOWN_E (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR) is

       NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
       IF NEW_LOC(2) < MAX_COL and NEW_LOC(3) < MAX_DEP
       then
         NEW_LOC(2) := NEW_LOC(2) + 1;
         NEW_LOC(3) := NEW_LOC(3) + 1;
         CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
       end if;
end CHECK_DOWN_E;

procedure CHECK_W (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR) is

       NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
       IF NEW_LOC(2) > 1 then
         NEW_LOC(2) := NEW_LOC(2) - 1;
         CK_STATE (NEW_LOC, N_ARRAY, CARD_COST, ROOT);
       end if;
end CHECK_W;

procedure CHECK_UP_W (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR) is
```

```
        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(2) > 1 and NEW_LOC(3) > 1 then
          NEW_LOC(2) := NEW_LOC(2) - 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_W;

procedure CHECK_DOWN_W (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(2) > 1 and NEW_LOC(3) < MAX_DEP then
          NEW_LOC(2) := NEW_LOC(2) - 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_W;

procedure CHECK_S (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          CK_STATE (NEW_LOC, N_ARRAY, CARD_COST, ROOT);
        end if;
end CHECK_S;

procedure CHECK_UP_S (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(3) > 1 then
```

```
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_S;

procedure CHECK_DOWN_S (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(3) < MAX_DEP
        then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_S;

procedure CHECK_SE (N_ARRAY : in out NODE_ARRAY;
                    ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) < MAX_COL
        then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) + 1;
          if ROOT.LOC(4) = 2 then
                NEW_LOC(4) := 3;
          else
                NEW_LOC(4) := 2;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_SE;

procedure CHECK_UP_SE (N_ARRAY  : in out NODE_ARRAY;
                       ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;
```

```
begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) < MAX_COL
        and NEW_LOC(3) > 1 then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) + 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          if ROOT.LOC(4) = 2 then
                NEW_LOC(4) := 3;
          else
                NEW_LOC(4) := 2;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_SE;

procedure CHECK_DOWN_SE (N_ARRAY  : in out NODE_ARRAY;
                         ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) < MAX_COL
        and NEW_LOC(3) < MAX_DEP then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) + 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          if ROOT.LOC(4) = 2 then
                NEW_LOC(4) := 3;
          else
                NEW_LOC(4) := 2;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_SE;

procedure CHECK_SW (N_ARRAY  : in out NODE_ARRAY;
                    ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) > 1 then
```

```
            NEW_LOC(1) := NEW_LOC(1) + 1;
            NEW_LOC(2) := NEW_LOC(2) - 1;
            if ROOT.LOC(4) = 3 then
                  NEW_LOC(4) := 4;
            else
                  NEW_LOC(4) := 3;
            end if;
            CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
          end if;
end CHECK_SW;

procedure CHECK_UP_SW (N_ARRAY  : in out NODE_ARRAY;
                       ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) > 1 and
        NEW_LOC(3) > 1 then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) - 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          if ROOT.LOC(4) = 3 then
                NEW_LOC(4) := 4;
          else
                NEW_LOC(4) := 3;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_SW;

procedure CHECK_DOWN_SW (N_ARRAY  : in out NODE_ARRAY;
                         ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) > 1 and
        NEW_LOC(3) < MAX_DEP then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) - 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          if ROOT.LOC(4) = 3 then
```

```
                NEW_LOC(4) := 4;
          else
                NEW_LOC(4) := 3;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_SW;

procedure CHECK_NW (N_ARRAY  : in out NODE_ARRAY;
                    ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) > 1 and NEW_LOC(2) > 1 then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          NEW_LOC(2) := NEW_LOC(2) - 1;
          if ROOT.LOC(4) = 1 then
                NEW_LOC(4) := 4;
          else
                NEW_LOC(4) := 1;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_NW;

procedure CHECK_UP_NW (N_ARRAY  : in out NODE_ARRAY;
                       ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) > 1 and NEW_LOC(2) > 1 and NEW_LOC(3)
        > 1 then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          NEW_LOC(2) := NEW_LOC(2) - 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          if ROOT.LOC(4) = 1 then
                NEW_LOC(4) := 4;
          else
                NEW_LOC(4) := 1;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
```

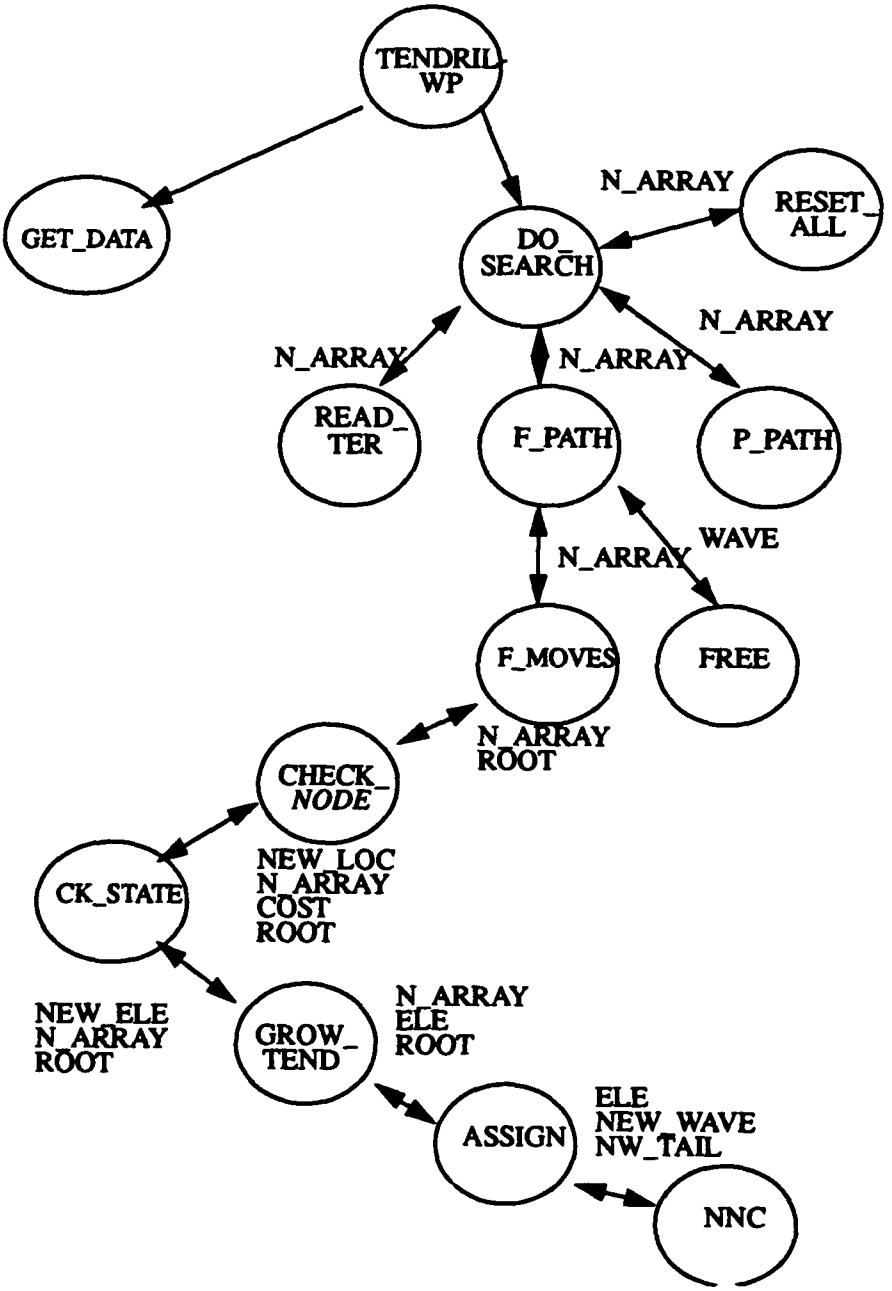
```
            end if;
        end CHECK_UP_NW;

        procedure CHECK_DOWN_NW (N_ARRAY  : in out NODE_ARRAY;
                                 ROOT     : in out LIST_PTR) is

            NEW_LOC : LOC_ARRAY := ROOT.LOC;

        begin
            IF NEW_LOC(1) > 1 and NEW_LOC(2) > 1 and NEW_LOC(3)
<
            MAX_DEP then
              NEW_LOC(1) := NEW_LOC(1) - 1;
              NEW_LOC(2) := NEW_LOC(2) - 1;
              NEW_LOC(3) := NEW_LOC(3) + 1;
              if ROOT.LOC(4) = 1 then
                    NEW_LOC(4) := 4;
              else
                    NEW_LOC(4) := 1;
              end if;
              CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
            end if;
        end CHECK_DOWN_NW;

    end THE_MOVE;
```

APPENDIX E (part 1)

Table 1: Data Dictionary for the TENDRILWP Search

PACKAGE PROCEDURE VARIABLE	TYPE
PATHWP GET_DATA FILE_NAME	 STRING (1..12)
NAME_LEN	INT_TYPE
CONT	CHARACTER := 'Y'
P_PATH NEXT_LOC	 LOC_ARRAY
F_MOVES HEADING	 INT_TYPE := ROOT.LOC (4)
F_PATH ROOT	 LIST_PTR := WAVE
DO_SEARCH N_ARRAY	 NODE_ARRAY (1..MAX_ROW, 1..MAX_COL,' 1..MAX_DEP, 1..MAX_HDG)

DATA FLOW DIAGRAM for the TENDRILWP SEARCH (part 2)



TENDRILWP SEARCH CODE (part 3)

```
--============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  TENDRILWP.ADA
--DESCRIPTION :  Main procedure for the Tendril search with
waypoints.
--CALLS        :  GET_DATA, and DO_SEARCH in the PATHWP package
--NOTES        :
--============================================================

with TEXT_IO, GLOBALS, PATHWP;
use TEXT_IO, GLOBALS, PATHWP;

procedure TENDRILWP is

begin
  GET_DATA;
  DO_SEARCH;
end TENDRILWP;
```

```
--============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  PATHWP.ADS
--DESCRIPTION :  Contains the major procedures for the Tendril
with waypoint
--             :  search
--CALLS        :
--NOTES        :
--============================================================

with TEXT_IO, GLOBALS, THE_MOVE, UNCHECKED_DEALLOCATION;
use TEXT_IO, GLOBALS, THE_MOVE;

package PATHWP is

  procedure DO_SEARCH;

  procedure GET_DATA;

  procedure READ_TER (N_ARRAY : in out NODE_ARRAY);

  procedure P_PATH (N_ARRAY : in out NODE_ARRAY);

end PATHWP;

--============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  PATHWP.ADB
--DESCRIPTION :  Package body for PATHWP package
--NOTES        :  Differences from PATH package:  GET_DATA
--             :  makes a
--            :  list of waypoints, P_PATH creates a list of
--             :  records in each path segment and writes to
--             :  file only (no screen output), RESET_ALL
--             :  resets global variables to initial settings
--             :  after each path segment is completed.
--============================================================

package body PATHWP is
```

```
    procedure GET_DATA is

--  Opens the terrain data file, reads in the array dimensions
--   and takes the starting and goal coordinates as input.

      FILE_NAME : STRING (1..12);
      NAME_LEN  : INT_TYPE;
      CONT      : CHARACTER := 'Y';

    begin
      put ("Enter the name of data file:  ");
      get_line (FILE_NAME, NAME_LEN);
      FILE_NAME ((NAME_LEN + 1)..12) := (others => ' ');
      FILE_NAME (9..12) := ".DAT";
      OPEN (DATA_FILE, MODE => IN_FILE, NAME => FILE_NAME);
      INT_IO.get (DATA_FILE, MAX_ROW);
      INT_IO.get (DATA_FILE, MAX_COL);
      INT_IO.get (DATA_FILE, MAX_DEP);
      INT_IO.get (DATA_FILE, MAX_HDG);
      NEW_LINE;
      WAVE_HEAD := new LIST;
      WAVE_TAIL := WAVE_HEAD;
      while CONT = 'y' or CONT = 'Y' loop
        put ("Enter the position row:  ");
        INT_IO.get (WAVE_TAIL.LOC(1));
        NEW_LINE;
        put ("Enter the position col:  ");
        INT_IO.get (WAVE_TAIL.LOC(2));
        NEW_LINE;
        put ("Enter the position dep:  ");
        INT_IO.get (WAVE_TAIL.LOC(3));
        NEW_LINE;
        put ("Enter the position hdg:  ");
        INT_IO.get (WAVE_TAIL.LOC(4));
        NEW_LINE;
        put ("Enter another position?");
        get (CONT);
        if CONT = 'y' or CONT = 'Y' then
          WAVE_TAIL.NEXT := new LIST;
          WAVE_TAIL := WAVE_TAIL.NEXT;
       end if;
       end loop;
```

```
    end GET_DATA;

    procedure READ_TER (N_ARRAY : in out NODE_ARRAY) is

--  Reads the terrain data from the data file and initializes
--  the N_ARRAY.

    begin
         for ROW in 1..MAX_ROW loop
           for COL in 1..MAX_COL loop
                for DEP in 1..MAX_DEP loop
                  for HDG in 1..MAX_HDG loop
               INT_IO.get (DATA_FILE, N_ARRAY(ROW, COL, DEP,
                                   HDG).STATE);
                  N_ARRAY(ROW, COL, DEP, HDG).TEND_LEN := 0;
                   N_ARRAY(ROW, COL, DEP, HDG).PARENT :=
(0,0,0,0);
                  end loop;
                end loop;
           end loop;
         end loop;
         close (DATA_FILE);
    end READ_TER;

    procedure P_PATH (N_ARRAY : in out NODE_ARRAY) is

--  Creates a list, THE_PATH and writes it to a file.

      NEXT_LOC  : LOC_ARRAY;

    begin
         if GOAL_FOUND and THE_PATH = null then
           THE_PATH := new LIST;
           THE_PATH.LOC := GOAL;
           loop
             NEXT_LOC := N_ARRAY(GOAL(1), GOAL(2), GOAL(3),
                                 GOAL(4)).PARENT;
             THE_PATH_CURRENT := new LIST;
             THE_PATH_CURRENT.LOC := NEXT_LOC;
             THE_PATH_CURRENT.NEXT := THE_PATH;
             THE_PATH := THE_PATH_CURRENT;
             GOAL := NEXT_LOC;
             exit when THE_PATH.LOC = START;
```

```
            end loop;
          elsif GOAL_FOUND and THE_PATH /= null then
            loop
              NEXT_LOC := N_ARRAY(GOAL(1), GOAL(2), GOAL(3),
                                  GOAL(4)).PARENT;
              THE_PATH_CURRENT := new LIST;
              THE_PATH_CURRENT.LOC := NEXT_LOC;
              THE_PATH_CURRENT.NEXT := THE_PATH;
              THE_PATH := THE_PATH_CURRENT;
              GOAL := NEXT_LOC;
              exit when THE_PATH.LOC = START;
            end loop;
          else
            put ("PATH NOT FOUND");
            new_line;
          end if;
          while THE_PATH /= null loop
            INT_IO.put (PATH_FILE, THE_PATH.LOC(1));
            INT_IO.put (PATH_FILE, THE_PATH.LOC(2));
            INT_IO.put (PATH_FILE, THE_PATH.LOC(3));
            INT_IO.put (PATH_FILE, THE_PATH.LOC(4));
            new_line (PATH_FILE);
            THE_PATH := THE_PATH.NEXT;
          end loop;
          put (PATH_FILE, "END PATH SEGMENT");
          new_line (PATH_FILE);
      end P_PATH;

      procedure F_MOVES (N_ARRAY : in out NODE_ARRAY;
                         ROOT    : in out LIST_PTR) is

--    Using the heading of the current node being processed,
--    F_MOVES determines the legal moves that can be made and
--    calls the CHECK_??? procedure in the THE_MOVE package.

            HEADING   : INT_TYPE := ROOT.LOC(4);

      begin
            case HEADING is
              when 1 =>
                    CHECK_UP_NW (N_ARRAY, ROOT);
                    CHECK_UP_N (N_ARRAY, ROOT);
                    CHECK_UP_NE (N_ARRAY, ROOT);
```

```
            CHECK_NW (N_ARRAY, ROOT);
            CHECK_N (N_ARRAY, ROOT);
            CHECK_NE (N_ARRAY, ROOT);
            CHECK_DOWN_NW (N_ARRAY, ROOT);
            CHECK_DOWN_N (N_ARRAY, ROOT);
            CHECK_DOWN_NE (N_ARRAY, ROOT);

        when 2 =>
            CHECK_UP_NE (N_ARRAY, ROOT);
            CHECK_UP_E (N_ARRAY, ROOT);
            CHECK_UP_SE (N_ARRAY, ROOT);
            CHECK_NE (N_ARRAY, ROOT);
            CHECK_E (N_ARRAY, ROOT);
            CHECK_SE (N_ARRAY, ROOT);
            CHECK_DOWN_NE (N_ARRAY, ROOT);
            CHECK_DOWN_E (N_ARRAY, ROOT);
            CHECK_DOWN_SE (N_ARRAY, ROOT);

        when 3 =>
            CHECK_UP_SE (N_ARRAY, ROOT);
            CHECK_UP_S (N_ARRAY, ROOT);
            CHECK_UP_SW (N_ARRAY, ROOT);
            CHECK_SE (N_ARRAY, ROOT);
            CHECK_S (N_ARRAY, ROOT);
            CHECK_SW (N_ARRAY, ROOT);
            CHECK_DOWN_SE (N_ARRAY, ROOT);
            CHECK_DOWN_S (N_ARRAY, ROOT);
            CHECK_DOWN_SW (N_ARRAY, ROOT);

        when 4 =>
            CHECK_UP_SW (N_ARRAY, ROOT);
            CHECK_UP_W (N_ARRAY, ROOT);
            CHECK_UP_NW (N_ARRAY, ROOT);
            CHECK_SW (N_ARRAY, ROOT);
            CHECK_W (N_ARRAY, ROOT);
            CHECK_NW (N_ARRAY, ROOT);
            CHECK_DOWN_SW (N_ARRAY, ROOT);
            CHECK_DOWN_W (N_ARRAY, ROOT);
            CHECK_DOWN_NW (N_ARRAY, ROOT);

        when others =>
            null;
```

```
            end case;
        end F_MOVES;

        procedure FREE is new UNCHECKED_DEALLOCATION (LIST,
    LIST_PTR);

    --  Clears old memory space.

        procedure F_PATH (N_ARRAY : in out NODE_ARRAY) is

    --  Processes the WAVE list in order calling the F_MOVE
    --  procedure.  Also reinitializes the WAVE list.

            ROOT : LIST_PTR := WAVE;

        begin
            while ROOT /= null loop
              F_MOVES (N_ARRAY, ROOT);
              ROOT := WAVE.NEXT;
              FREE (WAVE);
              WAVE := ROOT;
            end loop;
          FREE (WAVE);
            if GOAL_FOUND then
              return;
            end if;
            WAVE := NEW_WAVE;
            NEW_WAVE := null;
            LAST_NEW_WAVE := null;
        end F_PATH;

        procedure RESET_ALL (N_ARRAY : in out NODE_ARRAY) is

    --  Used to reset the various attributes in N_ARRAY changed
    --  during each path segment search.  This allows a path to
    --  go to a waypoint/goal and return using many of the nodes
    --  previously used in the outbound trip.

        begin
          GOAL_FOUND := FALSE;
          for ROW in 1..MAX_ROW loop
            for COL in 1..MAX_COL loop
              for DEP in 1..MAX_DEP loop
```

```
                for HDG in 1..MAX_HDG loop
                N_ARRAY(ROW,COL,DEP,HDG).PARENT := (others => 0);
                  N_ARRAY(ROW,COL,DEP,HDG).TEND_LEN := 0;
                end loop;
              end loop;
            end loop;
          end loop;
        end RESET_ALL;


        procedure DO_SEARCH is

    --  Calls the major procedures in the search and creates the
    --  N_ARRAY.

              N_ARRAY : NODE_ARRAY (1..MAX_ROW, 1..MAX_COL,
                                    1..MAX_DEP, 1..MAX_HDG);

        begin
           READ_TER (N_ARRAY);
           CREATE (PATH_FILE, NAME => "path.file");
           loop
             exit when WAVE_HEAD.NEXT = null;
             WAVE := new LIST;
             WAVE.LOC := WAVE_HEAD.LOC;
             START := WAVE_HEAD.LOC;
             GOAL := WAVE_HEAD.NEXT.LOC;
               while WAVE /= null and NOT GOAL_FOUND loop
                 F_PATH (N_ARRAY);
               end loop;
               P_PATH (N_ARRAY);
             RESET_ALL (N_ARRAY);
             WAVE_HEAD := WAVE_HEAD.NEXT;
           end loop;
           CLOSE (PATH_FILE);
        end DO_SEARCH;
    end PATHWP;
```

APPENDIX F (part 1)

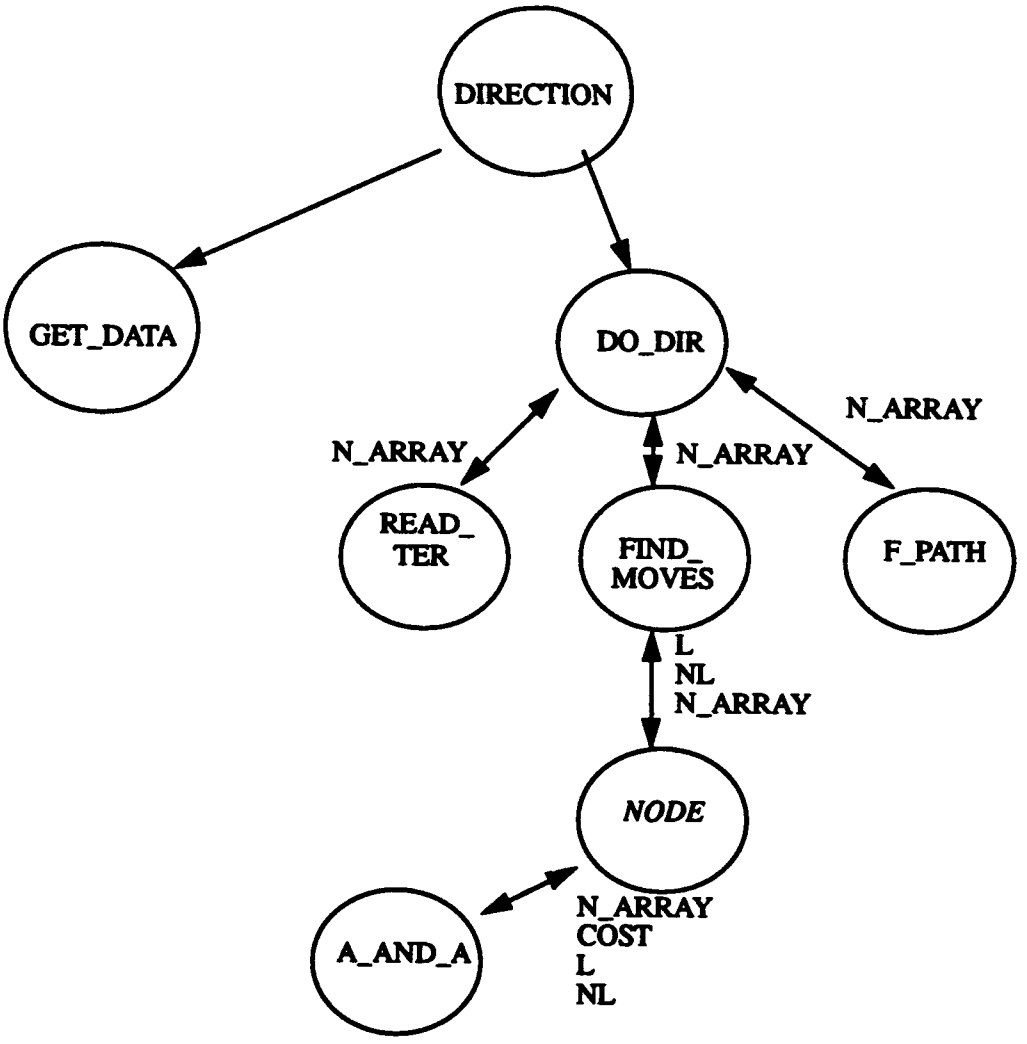
Table 1: Data Dictionary for the DIRECTION Search

PACKAGE PROCEDURE VARIABLE	TYPE
GLOBALS DATA_FILE	 FILE_TYPE
LOC_ARRAY	array (1..4) of INT_TYPE
LIST	record {LOC : LOC_ARRAY NEXT : LIST_PTR}
NODE	record {STATE : INT_TYPE DIST : INT_TYPE := 0 INC : INT_TYPE := 0 NEXT : LOC_ARRAY := (9,9,9,9)}
NODE_ARRAY	array (INT_TYPE range <>, INT_TYPE range <>, INT_TYPE range <>, INT_TYPE range <>) of NODE
START_TIME END_TIME T_TIME	TIME DURATION
MAX_ROW MAX_COL MAX_DEP MAX_HDG DIAG_COST := 120 CARD_COST := 100	INT_TYPE
ACTIVE TAIL CLEAR PATH	LIST_PTR := null
START GOAL	LOC_ARRAY
GOAL_FOUND	BOOLEAN := FALSE

Table 1: Data Dictionary for the DIRECTION Search

PACKAGE PROCEDURE VARIABLE	TYPE
B GET_DATA FILE_NAME	 STRING (1..12)
NAME_LEN	INT_TYPE
P_PATH P_TAIL	 LIST_PTR := null
FIND_MOVES HDG : ACTIVE_LOC (4)	 INT_TYPE
L NL	LOC_ARRAY := ACTIVE.LOC
FIND_PATH F_TAIL	 LIST_PTR := null
DO_DIR N_ARRAY	 NODE_ARRAY (1..MAX_ROW, 1..MAX_COL, 1..MAX_DEP, 1..MAX_HDG)

DATA FLOW DIAGRAM for the DIRECTION SEARCH (part 2)



DIRECTION SEARCH CODE (part 3)

```
--============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  DIRECTION.ADA
--DESCRIPTION  :  Main procedure for the Direction search
--CALLS        : GET_DATA and DO_DIR in the B package (like PATH
--             :  package).
--NOTES        :
--============================================================

with TEXT_IO, A, B;
use TEXT_IO, A, B;

procedure DIRECTION is

begin
  GET_DATA;
  DO_DIR;
end DIRECTION;
```

```
--============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  A.ADS
--DESCRIPTION  :  Global variables for the Direction search.
--CALLS        :
--NOTES        :
--============================================================

with TEXT_IO, CALENDAR;
use TEXT_IO, CALENDAR;

package A is

  subtype INT_TYPE is INTEGER;
  package INT_IO is new INTEGER_IO (INT_TYPE);
  package FLOATIO is new FLOAT_IO (FLOAT);
  use INT_IO, FLOATIO;

  type LOC_ARRAY is array (1..4) of INT_TYPE;

  type LIST;
  type LIST_PTR is access LIST;
  type LIST is
    record
           LOC  : LOC_ARRAY;
           NEXT : LIST_PTR;
    end record;

  ACTIVE : LIST_PTR;
  TAIL   : LIST_PTR;
  CLEAR  : LIST_PTR;
  PATH   : LIST_PTR;

  type NODE is
    record
           STATE   : INT_TYPE := 0;
           DIST    : INT_TYPE := 0;
           INC     : INT_TYPE := 0;
         NEXT    : LOC_ARRAY := (9,9,9,9);
    end record;
```

```
   type NODE_ARRAY is array (INT_TYPE range <>, INT_TYPE range
                             <>,INT_TYPE range <>, INT_TYPE
                             range <>) of NODE;

   DATA_FILE : FILE_TYPE;

   START_TIME : TIME;
   END_TIME   : TIME;

   T_TIME     : DURATION;

   MAX_ROW : INT_TYPE;
   MAX_COL : INT_TYPE;
   MAX_DEP : INT_TYPE;
   MAX_HDG : INT_TYPE := 4;

   DIAG_COST : INT_TYPE := 120;
   CARD_COST : INT_TYPE := 100;

   GOAL  : LOC_ARRAY;
   START : LOC_ARRAY;

end A;
```

```
--=============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  B.ADS
--DESCRIPTION :  This package is similar to the PATH package
in other programs.
--             :  It contains the major procedures for the
Direction search.
--CALLS        :
--NOTES        :
--=============================================================

with TEXT_IO, A, C, UNCHECKED_DEALLOCATION, CALENDAR;
use TEXT_IO, A, C, CALENDAR;

package B is

  procedure DO_DIR;

  procedure GET_DATA;

end B;

--=============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  B.ADB
--DESCRIPTION :  Package body for the B package
--=============================================================


package body B is

  procedure GET_DATA is

-- Opens the Data file containing terrain information and gets
-- the array dimensions.  It also takes as input the starting
-- and goal coordinates.

    FILE_NAME : STRING (1..12);
    NAME_LEN  : INT_TYPE;
```

```
  begin
    put ("Enter the name of data file:  ");
    get_line (FILE_NAME, NAME_LEN);
    FILE_NAME ((NAME_LEN + 1)..12) := (others => ' ');
    FILE_NAME (9..12) := ".DAT";
    OPEN (DATA_FILE, MODE => IN_FILE, NAME => FILE_NAME);
    INT_IO.get (DATA_FILE, MAX_ROW);
    INT_IO.get (DATA_FILE, MAX_COL);
    INT_IO.get (DATA_FILE, MAX_DEP);
    INT_IO.get (DATA_FILE, MAX_HDG);
    NEW_LINE;
    put ("Enter the starting row:  ");
    INT_IO.get (START(1));
    NEW_LINE;
    put ("Enter the starting col:  ");
    INT_IO.get (START(2));
    NEW_LINE;
    put ("Enter the starting dep:  ");
    INT_IO.get (START(3));
    NEW_LINE;
    put ("Enter the starting hdg:  ");
    INT_IO.get (START(4));
    NEW_LINE;
    put ("Enter the goal row:  ");
    INT_IO.get (GOAL(1));
    NEW_LINE;
    put ("Enter the goal col:  ");
    INT_IO.get (GOAL(2));
    NEW_LINE;
    put ("Enter the goal dep:  ");
    INT_IO.get (GOAL(3));
    NEW_LINE;
    put ("Enter the goal hdg:  ");
    INT_IO.get (GOAL(4));
    ACTIVE := new LIST;
    ACTIVE.LOC := GOAL;
  end GET_DATA;

    procedure READ_TER (N_ARRAY : in out NODE_ARRAY;
                        D_FILE  : in out FILE_TYPE) is

-- Reads in the terrain data from the file and initializes the
```

```
--  N_ARRAY.

    begin
            for ROW in 1..MAX_ROW loop
              for COL in 1..MAX_COL loop
                      for DEP in 1..MAX_DEP loop
                        for HDG in 1..MAX_HDG loop
                         INT_IO.get (D_FILE, N_ARRAY(ROW, COL,
DEP,
                                                HDG).STATE);
                        end loop;
                      end loop;
              end loop;
            end loop;
            close (D_FILE);
    end READ_TER;

    procedure FREE is new UNCHECKED_DEALLOCATION (LIST,
LIST_PTR);

--  Used to free old memory space.

    procedure FIND_MOVES (N_ARRAY : in out NODE_ARRAY) is

--  Using the REVERSE heading of the node being processed,
--  FIND_MOVES determines which nodes can legally move into
--  it.  This is OPPOSITE from the other search methods.  Again
--  procedures in the C package (just like THE_MOVE) are
--  called to process the individual nodes.

            HDG : INT_TYPE := ACTIVE.LOC(4);
            L   : LOC_ARRAY := ACTIVE.LOC;
            NL  : LOC_ARRAY := ACTIVE.LOC;

    begin
            while ACTIVE /= null loop
              HDG := ACTIVE.LOC(4);
              L := ACTIVE.LOC;
              NL := ACTIVE.LOC;
              case HDG is
                      when 1 =>
                        S (L, NL, N_ARRAY);
                        NL := L;
```

```
    US (L, NL, N_ARRAY);
    NL := L;
    DS (L, NL, N_ARRAY);
    NL := L;
    NL(4) := 4;
    SE (L, NL, N_ARRAY);
    NL(1..3) := L(1..3);
    U_SE (L, NL, N_ARRAY);
    NL(1..3) := L(1..3);
    DSE (L, NL, N_ARRAY);
    NL(1..3) := L(1..3);
    NL(4) := 2;
    SW (L, NL, N_ARRAY);
    NL(1..3) := L(1..3);
    USW (L, NL, N_ARRAY);
    NL(1..3) := L(1..3);
    DSW (L, NL, N_ARRAY);

  when 2 =>
    W (L, NL, N_ARRAY);
    NL := L;
    UW (L, NL, N_ARRAY);
    NL := L;
    DW (L, NL, N_ARRAY);
    NL := L;
    NL(4) := 3;
    NW (L, NL, N_ARRAY);
    NL(1..3) := L(1..3);
    UNW (L, NL, N_ARRAY);
    NL(1..3) := L(1..3);
    DNW (L, NL, N_ARRAY);
    NL := L;
    NL(4) := 1;
    SW (L, NL, N_ARRAY);
    NL(1..3) := L(1..3);
    USW (L, NL, N_ARRAY);
    NL(1..3) := L(1..3);
    DSW (L, NL, N_ARRAY);

  when 3 =>
    N (L, NL, N_ARRAY);
    NL := L;
    UN (L, NL, N_ARRAY);
```

```
            NL := L;
            DN (L, NL, N_ARRAY);
            NL := L;
            NL(4) := 2;
            NW (L, NL, N_ARRAY);
            NL(1..3) := L(1..3);
            UNW (L, NL, N_ARRAY);
            NL(1..3) := L(1..3);
            DNW (L, NL, N_ARRAY);
            NL := L;
            NL(4) := 4;
            NE (L, NL, N_ARRAY);
            NL(1..3) := L(1..3);
            UNE (L, NL, N_ARRAY);
            NL(1..3) := L(1..3);
            DNE (L, NL, N_ARRAY);

          when 4 =>
            E (L, NL, N_ARRAY);
            NL := L;
            UE (L, NL, N_ARRAY);
            NL := L;
            DE (L, NL, N_ARRAY);
            NL := L;
            NL(4) := 3;
            NE (L, NL, N_ARRAY);
            NL(1..3) := L(1..3);
            UNE (L, NL, N_ARRAY);
            NL(1..3) := L(1..3);
            DNE (L, NL, N_ARRAY);
            NL := L;
            NL(4) := 1;
            SE (L, NL, N_ARRAY);
            NL(1..3) := L(1..3);
            U_SE (L, NL, N_ARRAY);
            NL(1..3) := L(1..3);
            DSE (L, NL, N_ARRAY);

          when others =>
            null;

      end case;
      CLEAR := ACTIVE;
```

```
            ACTIVE := ACTIVE.NEXT;
          CLEAR.NEXT := null;
            FREE(CLEAR);
        end loop;
    end FIND_MOVES;

  procedure P_PATH is

--  Processes the PATH list for printing.

    P_TAIL : LIST_PTR := PATH;

  begin
    loop
      put ('(');
      INT_IO.put (P_TAIL.LOC(1));
      INT_IO.put (P_TAIL.LOC(2));
      INT_IO.put (P_TAIL.LOC(3));
      INT_IO.put (P_TAIL.LOC(4));
      put (')');
      NEW_LINE;
      exit when P_TAIL.LOC = GOAL;
      P_TAIL := P_TAIL.NEXT;
    end loop;
  end P_PATH;

  procedure FIND_PATH (N_ARRAY : in out NODE_ARRAY) is

--  Used to process the PATH list adding nodes as the vectors
--  are traced.

    F_TAIL          : LIST_PTR;

  begin
    F_TAIL := new LIST;
    F_TAIL.LOC := START;
    PATH := F_TAIL;
    loop
      exit when F_TAIL.LOC = GOAL;
      F_TAIL.NEXT := new LIST;
     F_TAIL.NEXT.LOC := N_ARRAY(F_TAIL.LOC(1),F_TAIL.LOC(2),
                                F_TAIL.LOC(3),F_TAIL.LOC(4)).NEXT;
      F_TAIL := F_TAIL.NEXT;
```

```
    end loop;
  end FIND_PATH;


  procedure DO_DIR is

-- Creates the N_ARRAY dynamically and calls the major
-- procedures for the search.  Some timing constructs are
-- added for evaluation purposes.

    N_ARRAY : NODE_ARRAY(1..MAX_ROW,1..MAX_COL,
                         1..MAX_DEP,1..MAX_HDG);

  begin
    START_TIME := CLOCK;
    READ_TER (N_ARRAY, DATA_FILE);
    FIND_MOVES (N_ARRAY);
    FIND_PATH (N_ARRAY);
    P_PATH;
    END_TIME := CLOCK;
    T_TIME := END_TIME - START_TIME;
    FLOATIO.put (FLOAT(T_TIME));
    put (" seconds.");
    NEW_LINE;
  end DO_DIR;

end B;
```

```
--============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  C.ADS
--DESCRIPTION  :  The package containing THE_MOVE type
--             :  procedures
--CALLS        :
--NOTES        :  Although similar to THE_MOVE in other
--             :  programs, the procedures are somewhat
--             :  different due to the "reverse" nature of the
--             :  search.
--============================================================

with TEXT_IO, A;
use TEXT_IO, A;

package C is

  procedure A_AND_A (L        : in out LOC_ARRAY;
                     NL       : in out LOC_ARRAY;
                     N_ARRAY  : in out NODE_ARRAY;
                     I        : in out INT_TYPE);

  procedure N (L       : in out LOC_ARRAY;
               NL      : in out LOC_ARRAY;
               N_ARRAY : in out NODE_ARRAY);

  procedure NE (L       : in out LOC_ARRAY;
                NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY);

  procedure E (L       : in out LOC_ARRAY;
               NL      : in out LOC_ARRAY;
               N_ARRAY : in out NODE_ARRAY);

  procedure SE (L       : in out LOC_ARRAY;
                NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY);

  procedure S (L       : in out LOC_ARRAY;
               NL      : in out LOC_ARRAY;
               N_ARRAY : in out NODE_ARRAY);
```

```
procedure SW (L        : in out LOC_ARRAY;
              NL       : in out LOC_ARRAY;
              N_ARRAY  : in out NODE_ARRAY);

procedure W (L        : in out LOC_ARRAY;
             NL       : in out LOC_ARRAY;
             N_ARRAY  : in out NODE_ARRAY);

procedure NW (L        : in out LOC_ARRAY;
              NL       : in out LOC_ARRAY;
              N_ARRAY  : in out NODE_ARRAY);

procedure UN (L        : in out LOC_ARRAY;
              NL       : in out LOC_ARRAY;
              N_ARRAY  : in out NODE_ARRAY);

procedure UNE (L        : in out LOC_ARRAY;
               NL       : in out LOC_ARRAY;
               N_ARRAY  : in out NODE_ARRAY);

procedure 'E (L        : in out LOC_ARRAY;
              NL       : in out LOC_ARRAY;
              N_ARRAY  : in out NODE_ARRAY);

procedure U_SE (L        : in out LOC_ARRAY;
                NL       : in out LOC_ARRAY;
                N_ARRAY  : in out NODE_ARRAY);

procedure US (L        : in out LOC_ARRAY;
              NL       : in out LOC_ARRAY;
              N_ARRAY  : in out NODE_ARRAY);

procedure USW (L        : in out LOC_ARRAY;
               NL       : in out LOC_ARRAY;
               N_ARRAY  : in out NODE_ARRAY);

procedure UW (L        : in out LOC_ARRAY;
              NL       : in out LOC_ARRAY;
              N_ARRAY  : in out NODE_ARRAY);

procedure UNW (L        : in out LOC_ARRAY;
               NL       : in out LOC_ARRAY;
```

```
                N_ARRAY : in out NODE_ARRAY);

  procedure DN (L       : in out LOC_ARRAY;
                NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY);

  procedure DNE (L       : in out LOC_ARRAY;
                 NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY);

  procedure DE (L       : in out LOC_ARRAY;
                NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY);

  procedure DSE (L       : in out LOC_ARRAY;
                 NL      : in out LOC_ARRAY;
                 N_ARRAY : in out NODE_ARRAY);

  procedure DS (L       : in out LOC_ARRAY;
                NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY);

  procedure DSW (L       : in out LOC_ARRAY;
                 NL      : in out LOC_ARRAY;
                 N_ARRAY : in out NODE_ARRAY);

  procedure DW (L       : in out LOC_ARRAY;
                NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY);

  procedure DNW (L       : in out LOC_ARRAY;
                 NL      : in out LOC_ARRAY;
                 N_ARRAY : in out NODE_ARRAY);

end C;

--=============================================================
--NAME        :  J. Bonsignore, Jr.
--DATE        :  22 Jan, 1991
--REVISED     :
--TITLE       :  C.ADS
--DESCRIPTION :  The package body for the C package
--=============================================================
```

```
package body C is

  procedure A_AND_A (L              : in out LOC_ARRAY;
                     NL             : in out LOC_ARRAY;
                     N_ARRAY        : in out NODE_ARRAY;
                     I              : in out INT_TYPE) is

-- Analyzes and assigns the legal nodes to the ACTIVE list and
--  sets the various attributes in the N_ARRAY node.

    begin
      if N_ARRAY(NL(1), NL(2), NL(3), NL(4)).STATE = 0 then
           if N_ARRAY(NL(1), NL(2), NL(3), NL(4)).DIST = 0
           then
             if ACTIVE.NEXT = null then
                    TAIL := new LIST;
                    TAIL.LOC := NL;
                    ACTIVE.NEXT := TAIL;
             else
                    TAIL.NEXT := new LIST;
                    TAIL := TAIL.NEXT;
                    TAIL.LOC := NL;
             end if;
             N_ARRAY(NL(1),NL(2),NL(3),NL(4)).NEXT := L;
             N_ARRAY(NL(1),NL(2),NL(3),NL(4)).INC := I;
             N_ARRAY(NL(1),NL(2),NL(3),NL(4)).DIST :=
              N_ARRAY(L(1),L(2),L(3),L(4)).DIST + I;
           end if;
      end if;
    end A_AND_A;


-- The following procedures all determine the coordinates for
--  the given move based on the current node coordinates.
--  Calls the A_AND_A procedure.

  procedure N (L        : in out LOC_ARRAY;
               NL       : in out LOC_ARRAY;
               N_ARRAY  : in out NODE_ARRAY) is

    begin
           if L(1) > 1 then
```

```
            NL(1) := NL(1) - 1;
            A_AND_A (L, NL, N_ARRAY, CARD_COST);
          end if;
  end N;

procedure NE (L       : in out LOC_ARRAY;
              NL      : in out LOC_ARRAY;
              N_ARRAY : in out NODE_ARRAY) is

  begin
          if L(1) > 1 and L(2) < MAX_COL then
            NL(1) := NL(1) - 1;
            NL(2) := NL(2) + 1;
            A_AND_A (L, NL, N_ARRAY, DIAG_COST);
          end if;
  end NE;

procedure E (L       : in out LOC_ARRAY;
             NL      : in out LOC_ARRAY;
             N_ARRAY : in out NODE_ARRAY) is

  begin
          if L(2) < MAX_COL then
            NL(2) := NL(2) + 1;
            A_AND_A (L, NL, N_ARRAY, CARD_COST);
          end if;
  end E;

procedure SE (L       : in out LOC_ARRAY;
              NL      : in out LOC_ARRAY;
              N_ARRAY : in out NODE_ARRAY) is

  begin
          if L(1) < MAX_ROW and L(2) < MAX_COL then
            NL(1) := NL(1) + 1;
            NL(2) := NL(2) + 1;
            A_AND_A (L, NL, N_ARRAY, DIAG_COST);
          end if;
  end SE;

procedure S (L       : in out LOC_ARRAY;
             NL      : in out LOC_ARRAY;
             N_ARRAY : in out NODE_ARRAY) is
```

```
  begin
        if L(1) < MAX_ROW then
          NL(1) := NL(1) + 1;
          A_AND_A (L, NL, N_ARRAY, CARD_COST);
        end if;
  end S;

procedure SW (L       : in out LOC_ARRAY;
              NL      : in out LOC_ARRAY;
              N_ARRAY : in out NODE_ARRAY) is

  begin
        if L(1) < MAX_ROW and L(2) > 1 then
          NL(1) := NL(1) + 1;
          NL(2) := NL(2) - 1;
          A_AND_A (L, NL, N_ARRAY, DIAG_COST);
        end if;
  end SW;

procedure W (L       : in out LOC_ARRAY;
             NL      : in out LOC_ARRAY;
             N_ARRAY : in out NODE_ARRAY) is

  begin
        if L(2) > 1 then
          NL(2) := NL(2) - 1;
          A_AND_A (L, NL, N_ARRAY, CARD_COST);
        end if;
  end W;

procedure NW (L       : in out LOC_ARRAY;
              NL      : in out LOC_ARRAY;
              N_ARRAY : in out NODE_ARRAY) is

  begin
        if L(1) > 1 and L(2) > 1 then
          NL(1) := NL(1) - 1;
          NL(2) := NL(2) - 1;
          A_AND_A (L, NL, N_ARRAY, DIAG_COST);
        end if;
  end NW;
```

```
procedure UN (L       : in out LOC_ARRAY;
              NL      : in out LOC_ARRAY;
              N_ARRAY : in out NODE_ARRAY) is

  begin
        if L(1) > 1 and L(3) > 1 then
          NL(1) := NL(1) - 1;
          NL(3) := NL(3) - 1;
          A_AND_A (L, NL, N_ARRAY, DIAG_COST);
        end if;
  end UN;

procedure UNE (L       : in out LOC_ARRAY;
               NL      : in out LOC_ARRAY;
               N_ARRAY : in out NODE_ARRAY) is

  begin
        if L(1) > 1 and L(2) < MAX_COL and L(3) > 1 then
          NL(1) := NL(1) - 1;
          NL(2) := NL(2) + 1;
          NL(3) := NL(3) - 1;
          A_AND_A (L, NL, N_ARRAY, DIAG_COST);
        end if;
  end UNE;

procedure UE (L       : in out LOC_ARRAY;
              NL      : in out LOC_ARRAY;
              N_ARRAY : in out NODE_ARRAY) is

  begin
        if L(2) < MAX_COL and L(3) > 1 then
          NL(2) := NL(2) + 1;
          NL(3) := NL(3) - 1;
          A_AND_A (L, NL, N_ARRAY, DIAG_COST);
        end if;
  end UE;

procedure U_SE (L       : in out LOC_ARRAY;
                NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY) is

  begin
        if L(1) < MAX_ROW and L(2) < MAX_COL and L(3) > 1
```

```
then
            NL(1) := NL(1) + 1;
            NL(2) := NL(2) + 1;
            NL(3) := NL(3) - 1;
            A_AND_A (L, NL, N_ARRAY, DIAG_COST);
          end if;
    end U_SE;

  procedure US (L       : in out LOC_ARRAY;
                NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY) is

    begin
          if L(1) < MAX_ROW and L(3) > 1 then
            NL(1) := NL(1) + 1;
            NL(3) := NL(3) - 1;
            A_AND_A (L, NL, N_ARRAY, DIAG_COST);
          end if;
    end US;

  procedure USW (L       : in out LOC_ARRAY;
                 NL      : in out LOC_ARRAY;
                 N_ARRAY : in out NODE_ARRAY) is

    begin
          if L(1) < MAX_ROW and L(2) > 1 and L(3) > 1 then
            NL(1) := NL(1) + 1;
            NL(2) := NL(2) - 1;
            NL(3) := NL(3) - 1;
            A_AND_A (L, NL, N_ARRAY, DIAG_COST);
          end if;
    end USW;

  procedure UW (L       : in out LOC_ARRAY;
                NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY) is

    begin
          if L(2) > 1 and L(3) > 1 then
            NL(2) := NL(2) - 1;
            NL(3) := NL(3) - 1;
            A_AND_A (L, NL, N_ARRAY, DIAG_COST);
          end if;
```

```
  end UW;

procedure UNW (L       : in out LOC_ARRAY;
               NL      : in out LOC_ARRAY;
               N_ARRAY : in out NODE_ARRAY) is

  begin
        if L(1) > 1 and L(2) > 1 and L(3) > 1 then
          NL(1) := NL(1) - 1;
          NL(2) := NL(2) - 1;
          NL(3) := NL(3) - 1;
          A_AND_A (L, NL, N_ARRAY, DIAG_COST);
        end if;
  end UNW;

procedure DN (L       : in out LOC_ARRAY;
              NL      : in out LOC_ARRAY;
              N_ARRAY : in out NODE_ARRAY) is

  begin
        if L(1) > 1 and L(3) < MAX_DEP then
          NL(1) := NL(1) - 1;
          NL(3) := NL(3) + 1;
          A_AND_A (L, NL, N_ARRAY, DIAG_COST);
        end if;
  end DN;

procedure DNE (L       : in out LOC_ARRAY;
                  NL      : in out LOC_ARRAY;
                  N_ARRAY : in out NODE_ARRAY) is

  begin
        if L(1) > 1 and L(2) < MAX_COL and L(3) < MAX_DEP
        then
          NL(1) := NL(1) - 1;
          NL(2) := NL(2) + 1;
          NL(3) := NL(3) + 1;
          A_AND_A (L, NL, N_ARRAY, DIAG_COST);
        end if;
  end DNE;


procedure DE (L       : in out LOC_ARRAY;
```

```
                NL       : in out LOC_ARRAY;
                N_ARRAY  : in out NODE_ARRAY) is

  begin
         if L(2) < MAX_COL and L(3) < MAX_DEP then
           NL(2) := NL(2) + 1;
           NL(3) := NL(3) + 1;
           A_AND_A (L, NL, N_ARRAY, DIAG_COST);
         end if;
  end DE;


procedure DSE (L        : in out LOC_ARRAY;
               NL       : in out LOC_ARRAY;
               N_ARRAY  : in out NODE_ARRAY) is

  begin
         if L(1) < MAX_ROW and L(2) < MAX_COL and L(3) <
                   MAX_DEP then
           NL(1) := NL(1) + 1;
           NL(2) := NL(2) + 1;
           NL(3) := NL(3) + 1;
           A_AND_A (L, NL, N_ARRAY, DIAG_COST);
         end if;
  end DSE;

procedure DS (L        : in out LOC_ARRAY;
              NL       : in out LOC_ARRAY;
              N_ARRAY  : in out NODE_ARRAY) is

  begin
         if L(1) < MAX_ROW and L(3) < MAX_DEP then
           NL(1) := NL(1) + 1;
           NL(3) := NL(3) + 1;
           A_AND_A (L, NL, N_ARRAY, DIAG_COST);
         end if;
  end DS;

procedure DSW (L        : in out LOC_ARRAY;
               NL       : in out LOC_ARRAY;
               N_ARRAY  : in out NODE_ARRAY) is

  begin
```

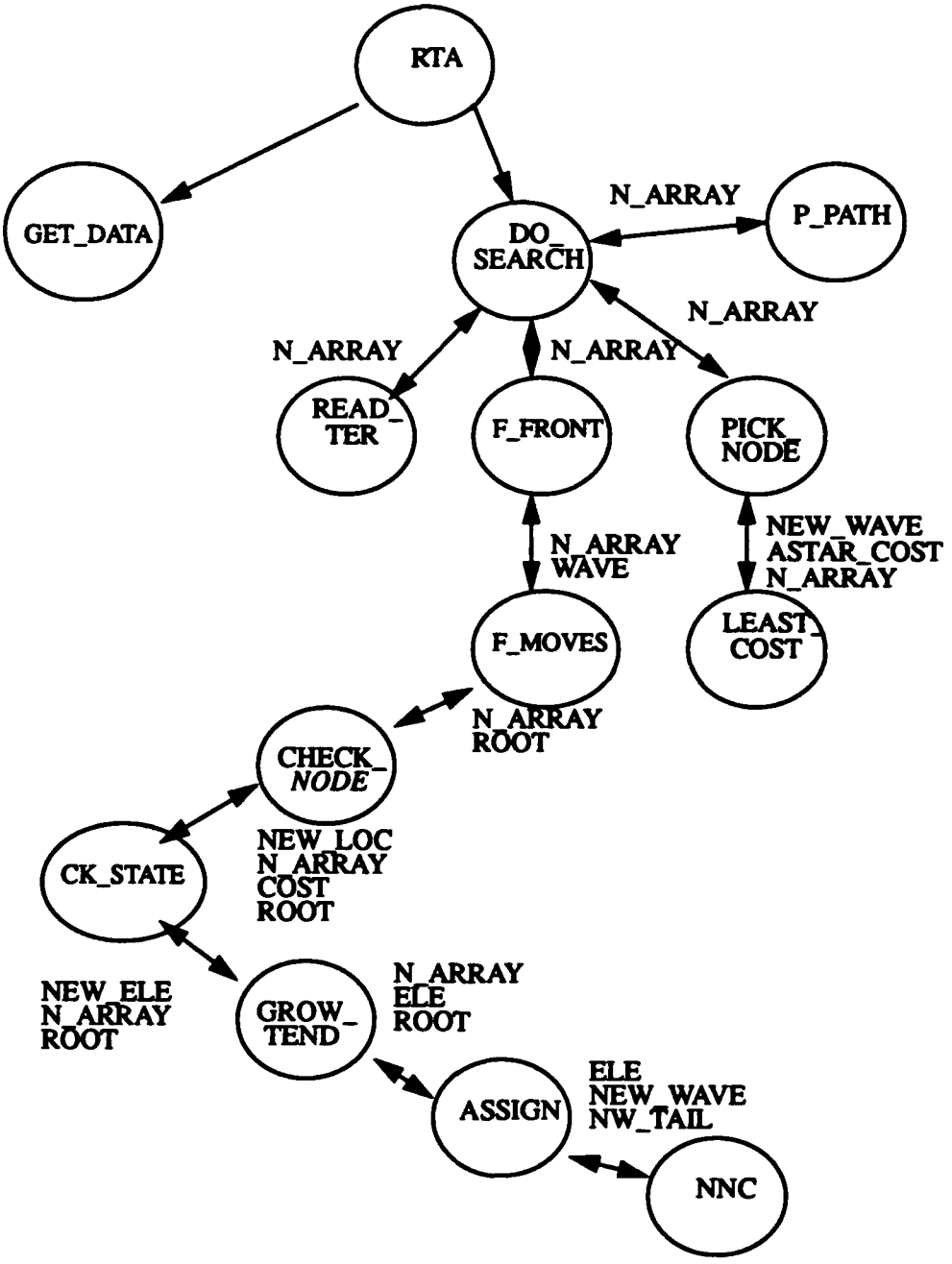
```
            if L(1) < MAX_ROW and L(2) > 1 and L(3) < MAX_DEP
then
              NL(1) := NL(1) + 1;
              NL(2) := NL(2) - 1;
              NL(3) := NL(3) + 1;
              A_AND_A (L, NL, N_ARRAY, DIAG_COST);
            end if;
    end DSW;

  procedure DW (L       : in out LOC_ARRAY;
                NL      : in out LOC_ARRAY;
                N_ARRAY : in out NODE_ARRAY) is

    begin
            if L(2) > 1 and L(3) < MAX_DEP then
              NL(2) := NL(2) - 1;
              NL(3) := NL(3) + 1;
              A_AND_A (L, NL, N_ARRAY, DIAG_COST);
            end if;
    end DW;

  procedure DNW (L       : in out LOC_ARRAY;
                 NL      : in out LOC_ARRAY;
                 N_ARRAY : in out NODE_ARRAY) is

    begin
            if L(1) > 1 and L(2) > 1 and L(3) < MAX_DEP then
              NL(1) := NL(1) - 1;
              NL(2) := NL(2) - 1;
              NL(3) := NL(3) + 1;
              A_AND_A (L, NL, N_ARRAY, DIAG_COST);
            end if;
    end DNW;

end C;
```

APPENDIX G (part 1)

Table 1: Data Dictionary for the RTA* Search

PACKAGE PROCEDURE VARIABLE	TYPE
PATHWP GET_DATA FILE_NAME	 STRING (1..12)
NAME_LEN	INT_TYPE
P_PATH NEXT_LOC	 LOC_ARRAY
F_MOVES HEADING	 INT_TYPE := ROOT.LOC (4)
PICK_NODE ASTAR_COST	 INTEGER := 999
LEAST_COST ROW_COST COL_COST DEP_COST HDG_COST NEW_COST	 INTEGER := 0
DO_SEARCH N_ARRAY	 NODE_ARRAY (1..MAX_ROW, 1..MAX_COL,' 1..MAX_DEP, 1..MAX_HDG)

DATA FLOW DIAGRAM for the RTA* SEARCH (part 2)



REAL-TIME A* SEARCH CODE (part 3)

```
--============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  RTA.ADA
--DESCRIPTION  :  Main procedure for the RTA* search
--CALLS        :  GET_DATA and DO_SEARCH in another PATH
--             :package for
--             :  the RTA.
--NOTES        :
--============================================================

with TEXT_IO, GLOBALS, PATH;
use TEXT_IO, GLOBALS, PATH;

procedure RTA is

begin
  GET_DATA;
  DO_SEARCH;
end RTA;
```

```
--============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  PATH.ADS
--DESCRIPTION  :  Contains the major procedures for the RTA
--             :  search
--CALLS        :
--NOTES        :
--
--============================================================

with TEXT_IO, GLOBALS, THE_MOVE, UNCHECKED_DEALLOCATION;
use TEXT_IO, GLOBALS, THE_MOVE;

package PATH is

  procedure DO_SEARCH;

  procedure GET_DATA;

  procedure READ_TER (N_ARRAY : in out NODE_ARRAY);

  procedure P_PATH (N_ARRAY : in out NODE_ARRAY);

end PATH;

--============================================================

--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  PATH.ADB
--DESCRIPTION  :  Package body for the Path package used for
--             :  the RTA search.
--============================================================

package body PATH is

    procedure GET_DATA is

--  SAME AS OTHER PATH PACKAGE.
```

```
  FILE_NAME : STRING (1..12);
  NAME_LEN  : INT_TYPE;

begin
  put ("Enter the name of data file:  ");
  get_line (FILE_NAME, NAME_LEN);
  FILE_NAME ((NAME_LEN + 1)..12) := (others => ' ');
  FILE_NAME (9..12) := ".DAT";
  OPEN (DATA_FILE, MODE => IN_FILE, NAME => FILE_NAME);
  INT_IO.get (DATA_FILE, MAX_ROW);
  INT_IO.get (DATA_FILE, MAX_COL);
  INT_IO.get (DATA_FILE, MAX_DEP);
  INT_IO.get (DATA_FILE, MAX_HDG);
  NEW_LINE;
  put ("Enter the starting row:  ");
  INT_IO.get (START(1));
  NEW_LINE;
  put ("Enter the starting col:  ");
  INT_IO.get (START(2));
  NEW_LINE;
  put ("Enter the starting dep:  ");
  INT_IO.get (START(3));
  NEW_LINE;
  put ("Enter the starting hdg:  ");
  INT_IO.get (START(4));
  NEW_LINE;
  put ("Enter the goal row:  ");
  INT_IO.get (GOAL(1));
  NEW_LINE;
  put ("Enter the goal col:  ");
  INT_IO.get (GOAL(2));
  NEW_LINE;
  put ("Enter the goal dep:  ");
  INT_IO.get (GOAL(3));
  NEW_LINE;
  put ("Enter the goal hdg:  ");
  INT_IO.get (GOAL(4));
  WAVE := new LIST;
  WAVE.LOC := START;
  WAVE.INC := 0;
end GET_DATA;

procedure READ_TER (N_ARRAY : in out NODE_ARRAY) is
```

```
--   SAME AS OTHER PATH PACKAGE.

     begin
          for ROW in 1..MAX_ROW loop
            for COL in 1..MAX_COL loop
                 for DEP in 1..MAX_DEP loop
                   for HDG in 1..MAX_HDG loop
                    INT_IO.get (DATA_FILE, N_ARRAY(ROW, COL,
                                                   DEP, HDG).STATE);
                   N_ARRAY(ROW, COL, DEP, HDG).TEND_LEN := 0;
                    N_ARRAY(ROW, COL, DEP, HDG).PARENT :=
                                                   (0,0,0,0);
                   end loop;
                 end loop;
            end loop;
          end loop;
          close (DATA_FILE);
     end READ_TER;

     procedure P_PATH (N_ARRAY : in out NODE_ARRAY) is

--   SAME AS OTHER PATH PACKAGE.

          NEXT_LOC : LOC_ARRAY;

     begin
          if GOAL_FOUND then
            put ('(');
            INT_IO.put (GOAL(1));
            INT_IO.put (GOAL(2));
            INT_IO.put (GOAL(3));
            INT_IO.put (GOAL(4));
            put (')');
            new_line;
            NEXT_LOC := N_ARRAY(GOAL(1), GOAL(2), GOAL(3),
                                GOAL(4)).PARENT;
            while NEXT_LOC /= START loop
                 put ('(');
                 INT_IO.put (NEXT_LOC(1));
                 INT_IO.put (NEXT_LOC(2));
                 INT_IO.put (NEXT_LOC(3));
                 INT_IO.put (NEXT_LOC(4));
```

```
                put (')');
              NEXT_LOC := N_ARRAY(NEXT_LOC(1), NEXT_LOC(2),
                                  NEXT_LOC(3),
                NEXT_LOC(4)).PARENT;
                NEW_LINE;
            end loop;
            put ('(');
            INT_IO.put (START(1));
            INT_IO.put (START(2));
            INT_IO.put (START(3));
            INT_IO.put (START(4));
            put (')');
            new_line;
            INT_IO.put (N_ARRAY (GOAL(1), GOAL(2), GOAL(3),
                                 GOAL(4)).TEND_LEN);
            new_line;
          else
            put ("PATH NOT FOUND");
            new_line;
          end if;
    end P_PATH;

    procedure F_MOVES (N_ARRAY : in out NODE_ARRAY;
                       ROOT    : in out LIST_PTR) is

-- SAME AS OTHER PATH PACKAGE.

          HEADING   : INT_TYPE := ROOT.LOC(4);

    begin
          case HEADING is
            when 1 =>
                  CHECK_UP_NW (N_ARRAY, ROOT);
                  CHECK_UP_N (N_ARRAY, ROOT);
                  CHECK_UP_NE (N_ARRAY, ROOT);
                  CHECK_NW (N_ARRAY, ROOT);
                  CHECK_N (N_ARRAY, ROOT);
                  CHECK_NE (N_ARRAY, ROOT);
                  CHECK_DOWN_NW (N_ARRAY, ROOT);
                  CHECK_DOWN_N (N_ARRAY, ROOT);
                  CHECK_DOWN_NE (N_ARRAY, ROOT);

            when 2 =>
```

```
                    CHECK_UP_NE (N_ARRAY, ROOT);
                    CHECK_UP_E (N_ARRAY, ROOT);
                    CHECK_UP_SE (N_ARRAY, ROOT);
                    CHECK_NE (N_ARRAY, ROOT);
                    CHECK_E (N_ARRAY, ROOT);
                    CHECK_SE (N_ARRAY, ROOT);
                    CHECK_DOWN_NE (N_ARRAY, ROOT);
                    CHECK_DOWN_E (N_ARRAY, ROOT);
                    CHECK_DOWN_SE (N_ARRAY, ROOT);

                when 3 =>
                    CHECK_UP_SE (N_ARRAY, ROOT);
                    CHECK_UP_S (N_ARRAY, ROOT);
                    CHECK_UP_SW (N_ARRAY, ROOT);
                    CHECK_SE (N_ARRAY, ROOT);
                    CHECK_S (N_ARRAY, ROOT);
                    CHECK_SW (N_ARRAY, ROOT);
                    CHECK_DOWN_SE (N_ARRAY, ROOT);
                    CHECK_DOWN_S (N_ARRAY, ROOT);
                    CHECK_DOWN_SW (N_ARRAY, ROOT);

                when 4 =>
                    CHECK_UP_SW (N_ARRAY, ROOT);
                    CHECK_UP_W (N_ARRAY, ROOT);
                    CHECK_UP_NW (N_ARRAY, ROOT);
                    CHECK_SW (N_ARRAY, ROOT);
                    CHECK_W (N_ARRAY, ROOT);
                    CHECK_NW (N_ARRAY, ROOT);
                    CHECK_DOWN_SW (N_ARRAY, ROOT);
                    CHECK_DOWN_W (N_ARRAY, ROOT);
                    CHECK_DOWN_NW (N_ARRAY, ROOT);

                when others =>
                    null;

            end case;
        end F_MOVES;

        procedure FREE is new UNCHECKED_DEALLOCATION (LIST,
                                                      LIST_PTR);

    -- SAME AS OTHER PATH PACKAGE.
```

```
    procedure F_FRONT (N_ARRAY : in out NODE_ARRAY) is

--  This procedure makes two calls to the F_MOVES procedure.
--  Each call extends the search depth an additional node
--  distance.  For further frontier nodes subsequent calls to
--  F_MOVES can be made.

    begin
      F_MOVES (N_ARRAY, WAVE);
      WAVE := NEW_WAVE;
      NEW_WAVE := null;
      while WAVE /= null loop
        TRASH := WAVE;
        F_MOVES (N_ARRAY, WAVE);
        WAVE := WAVE.NEXT;
        FREE (TRASH);
      end loop;
    end F_FRONT;

    procedure PICK_NODE (N_ARRAY : in out NODE_ARRAY) is

--  Estimates the cost to reach the GOAL from the frontier
--  nodes and picks the least cost frontier node for further
--  expansion.  This is currently working in a limited manner.
--  Absolute values must be used during the substraction
--  of the GOAL from the LOC.

      ASTAR_COST : INTEGER := 999;

    procedure LEAST_COST (LOC        : in out LIST_PTR;
                          ASTAR_COST : in out INTEGER;
                          N_ARRAY    : in NODE_ARRAY) is

--  Estimates the cost to the GOAL and assigns that nodes
--  coordinates to ASTAR_
--  COST if it is less then previously processed nodes cost.

      ROW_COST    : INTEGER := 0;
      COL_COST    : INTEGER := 0;
      DEP_COST    : INTEGER := 0;
      HDG_COST    : INTEGER := 0;
      NEW_COST    : INTEGER := 0;
```

```
    begin
      ROW_COST := GOAL(1) - LOC.LOC(1);
      COL_COST := GOAL(2) - LOC.LOC(2);
      DEP_COST := GOAL(3) - LOC.LOC(3);
      HDG_COST := GOAL(4) - LOC.LOC(4);
      NEW_COST := ROW_COST + COL_COST + DEP_COST + HDG_COST;
--                  + N_ARRAY(LOC.LOC(1), LOC.LOC(2),
--                            LOC.LOC(3), LOC.LOC(4)).TEND_LEN;
      if NEW_COST < ASTAR_COST then
        ASTAR_COST := NEW_COST;
        WAVE := new LIST;
        WAVE.LOC := NEW_WAVE.LOC;
      end if;
    end LEAST_COST;


    begin
      while NEW_WAVE /= null loop
        LEAST_COST (NEW_WAVE, ASTAR_COST, N_ARRAY);
        if NEW_WAVE.LOC = GOAL then
          GOAL_FOUND := TRUE;
        end if;
        exit when GOAL_FOUND;
        NEW_WAVE := NEW_WAVE.NEXT;
      end loop;
      P_NODE (WAVE);
    end PICK_NODE;

    procedure DO_SEARCH is

-- Calls the major procedures in the search and creates the
--  N_ARRAY.

      N_ARRAY : NODE_ARRAY (1..MAX_ROW, 1..MAX_COL,
                            1..MAX_DEP, 1..MAX_HDG);

    begin
      READ_TER (N_ARRAY);
      while not GOAL_FOUND loop
        F_FRONT (N_ARRAY);
        PICK_NODE (N_ARRAY);
        while NEW_WAVE /= null loop
          TRASH := NEW_WAVE;
```

```
            NEW_WAVE := NEW_WAVE.NEXT;
            FREE (TRASH);
          end loop;
          NEW_WAVE := null;
        end loop;
        GOAL := WAVE.LOC;  -- Just for testing purposes!
        GOAL_FOUND := TRUE; -- Just for testing purposes!
        P_PATH (N_ARRAY);
      end DO_SEARCH;

end PATH;
```

```
--=============================================================
--NAME         :  J. Bonsignore, Jr.
--DATE         :  22 Jan, 1991
--REVISED      :
--TITLE        :  THE_MOVE.ADS
--DESCRIPTION  :  Package containing the procedures for
--              :  individual node processing.
--CALLS        :
--NOTES        :
--=============================================================
with TEXT_IO, GLOBALS;
use TEXT_IO, GLOBALS;

package THE_MOVE is

  procedure NNC (ELEMENT : in LIST_PTR;
                 HEAD    : in out LIST_PTR;
                 TAIL    : in out LIST_PTR);

  procedure GROW_TEND (ELE     : in out LIST_PTR;
                       N_ARRAY : in out NODE_ARRAY;
                       ROOT    : in out LIST_PTR);

  procedure CK_STATE (NEW_LOC  : in out LOC_ARRAY;
                      N_ARRAY  : in out NODE_ARRAY;
                      NEW_INC  : in out INT_TYPE;
                      ROOT     : in out LIST_PTR);

  procedure CHECK_N (N_ARRAY : in out NODE_ARRAY;
                     ROOT    : in out LIST_PTR);

  procedure CHECK_UP_N (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR);

  procedure CHECK_DOWN_N  (N_ARRAY : in out NODE_ARRAY;
                           ROOT    : in out LIST_PTR);

  procedure CHECK_NE (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR);

  procedure CHECK_UP_NE (N_ARRAY : in out NODE_ARRAY;
                         ROOT    : in out LIST_PTR);
```

```
procedure CHECK_DOWN_NE (N_ARRAY : in out NODE_ARRAY;
                         ROOT    : in out LIST_PTR);

procedure CHECK_E (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR);

procedure CHECK_UP_E (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR);

procedure CHECK_DOWN_E (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR);

procedure CHECK_SE (N_ARRAY : in out NODE_ARRAY;
                    ROOT    : in out LIST_PTR);

procedure CHECK_UP_SE (N_ARRAY : in out NODE_ARRAY;
                       ROOT    : in out LIST_PTR);

procedure CHECK_DOWN_SE (N_ARRAY : in out NODE_ARRAY;
                                   ROOT    : in out LIST_PTR);

procedure CHECK_S (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR);

procedure CHECK_UP_S (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR);

procedure CHECK_DOWN_S (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR);

procedure CHECK_SW (N_ARRAY : in out NODE_ARRAY;
                    ROOT    : in out LIST_PTR);

procedure CHECK_UP_SW (N_ARRAY : in out NODE_ARRAY;
                       ROOT    : in out LIST_PTR);

procedure CHECK_DOWN_SW (N_ARRAY : in out NODE_ARRAY;
                         ROOT    : in out LIST_PTR);

procedure CHECK_W (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR);

procedure CHECK_UP_W (N_ARRAY : in out NODE_ARRAY;
```

```
                        ROOT    : in out LIST_PTR);

  procedure CHECK_DOWN_W (N_ARRAY : in out NODE_ARRAY;
                          ROOT    : in out LIST_PTR);

  procedure CHECK_NW (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR);

  procedure CHECK_UP_NW (N_ARRAY : in out NODE_ARRAY;
                         ROOT    : in out LIST_PTR);

  procedure CHECK_DOWN_NW (N_ARRAY : in out NODE_ARRAY;
                           ROOT    : in out LIST_PTR);

end THE_MOVE;

--============================================================
--NAME        :  J. Bonsignore, Jr.
--DATE        :  22 Jan, 1991
--REVISED     :
--TITLE       :  THE_MOVE.ADB
--DESCRIPTION :  Package body for THE_MOVE package.
--============================================================

with TEXT_IO, GLOBALS;
use TEXT_IO, GLOBALS;

package body THE_MOVE is

    procedure NNC (ELEMENT  : in LIST_PTR;
                   HEAD     : in out LIST_PTR;
                   TAIL     : in out LIST_PTR) is

--  Creates and maintains lists.

           begin
             if HEAD = null then
                    HEAD := ELEMENT;
                    TAIL := ELEMENT;
             else
                    TAIL.NEXT := ELEMENT;
                    TAIL := TAIL.NEXT;
             end if;
```

```
        end NNC;

    procedure GROW_TEND (ELE      : in out LIST_PTR;
                         N_ARRAY : in out NODE_ARRAY;
                         ROOT     : in out LIST_PTR) is

--  Expands the search similar to that in the Tendril search.

        procedure ASSIGN (N_ARRAY : in out NODE_ARRAY;
                          ELE      : in out LIST_PTR;
                          ROOT     : in out LIST_PTR) is

        begin
        if N_ARRAY(ELE.LOC(1),ELE.LOC(2),ELE.LOC(3),
                   ELE.LOC(4)).TEND_LEN = 0 then
          NNC (ELE, NEW_WAVE, NW_TAIL);
        end if;
        N_ARRAY(ELE.LOC(1),ELE.LOC(2),ELE.LOC(3),
                ELE.LOC(4)).PARENT := ROOT.LOC;
        N_ARRAY(ELE.LOC(1),ELE.LOC(2),ELE.LOC(3),
                ELE.LOC(4)).TEND_LEN := ELE.INC +
        N_ARRAY(ROOT.LOC(1),ROOT.LOC(2),ROOT.LOC(3),
                ROOT.LOC(4)).TEND_LEN;
        if ELE.LOC = GOAL then
          GOAL_FOUND := TRUE;
        end if;
        end ASSIGN;

    begin
        if N_ARRAY(ELE.LOC(1),ELE.LOC(2),ELE.LOC(3),
                   ELE.LOC(4)).TEND_LEN = 0 then
          ASSIGN (N_ARRAY, ELE, ROOT);
        elsif N_ARRAY(ELE.LOC(1),ELE.LOC(2),ELE.LOC(3),
                      ELE.LOC(4)).TEND_LEN >
            (N_ARRAY(ROOT.LOC(1),ROOT.LOC(2),ROOT.LOC(3),
                     ROOT.LOC(4)).TEND_LEN + ELE.INC) then
          ASSIGN (N_ARRAY, ELE, ROOT);
        end if;
    end GROW_TEND;

    procedure CK_STATE (NEW_LOC   : in out LOC_ARRAY;
                        N_ARRAY   : in out NODE_ARRAY;
                        NEW_INC   : in out INT_TYPE;
```

```
                        ROOT       : in out LIST_PTR) is

--  Similar to CK_STATE in the Tendril search.

        NEW_ELE : LIST_PTR;

    begin
        if N_ARRAY(NEW_LOC(1), NEW_LOC(2), NEW_LOC(3),
                   NEW_LOC(4)).STATE = 0 then
          NEW_ELE := new LIST;
          NEW_ELE.LOC := NEW_LOC;
          NEW_ELE.INC := NEW_INC;
          GROW_TEND (NEW_ELE, N_ARRAY, ROOT);
        end if;
    end CK_STATE;

--  Following procedures are similar to those in the Tendril
--  search's THE_MOVE
--  package.

    procedure CHECK_N (N_ARRAY : in out NODE_ARRAY;
                       ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

    begin
        IF NEW_LOC(1) > 1 then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          CK_STATE (NEW_LOC, N_ARRAY, CARD_COST, ROOT);
        end if;
    end CHECK_N;

    procedure CHECK_UP_N (N_ARRAY : in out NODE_ARRAY;
                          ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

    begin
        IF NEW_LOC(1) > 1 and NEW_LOC(3) > 1 then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
```

```
end CHECK_UP_N;

procedure CHECK_DOWN_N (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR) is

       NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
       IF NEW_LOC(1) > 1 and NEW_LOC(3) < MAX_DEP then
         NEW_LOC(1) := NEW_LOC(1) - 1;
         NEW_LOC(3) := NEW_LOC(3) + 1;
         CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
       end if;
end CHECK_DOWN_N;

procedure CHECK_NE (N_ARRAY  : in out NODE_ARRAY;
                    ROOT     : in out LIST_PTR) is

       NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
       if NEW_LOC(1) > 1 and NEW_LOC(2) < MAX_COL then
         NEW_LOC(1) := NEW_LOC(1) - 1;
         NEW_LOC(2) := NEW_LOC(2) + 1;
         if ROOT.LOC(4) = 1 then
                NEW_LOC(4) := 2;
         else
                NEW_LOC(4) := 1;
         end if;
         CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
       end if;
end  CHECK_NE;

procedure CHECK_UP_NE (N_ARRAY  : in out NODE_ARRAY;
                       ROOT     : in out LIST_PTR) is

       NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
       IF NEW_LOC(1) > 1 and NEW_LOC(2) < MAX_COL and
       NEW_LOC(3) > 1 then
         NEW_LOC(1) := NEW_LOC(1) - 1;
         NEW_LOC(2) := NEW_LOC(2) + 1;
```

```
          NEW_LOC(3) := NEW_LOC(3) - 1;
          if ROOT.LOC(4) = 1 then
                 NEW_LOC(4) := 2;
          else
                 NEW_LOC(4) := 1;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_NE;

procedure CHECK_DOWN_NE (N_ARRAY  : in out NODE_ARRAY;
                         ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) > 1 and NEW_LOC(2) < MAX_COL and
        NEW_LOC(3) < MAX_DEP then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          NEW_LOC(2) := NEW_LOC(2) + 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          if ROOT.LOC(4) = 1 then
                 NEW_LOC(4) := 2;
          else
                 NEW_LOC(4) := 1;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_NE;

procedure CHECK_E (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(2) < MAX_COL then
          NEW_LOC(2) := NEW_LOC(2) + 1;
          CK_STATE (NEW_LOC, N_ARRAY, CARD_COST, ROOT);
        end if;
end CHECK_E;

procedure CHECK_UP_E (N_ARRAY : in out NODE_ARRAY;
```

```
                        ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(2) < MAX_COL and NEW_LOC(3) > 1 then
          NEW_LOC(2) := NEW_LOC(2) + 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_E;

procedure CHECK_DOWN_E (N_ARRAY : in out NODE_ARRAY;
                        ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(2) < MAX_COL and NEW_LOC(3) < MAX_DEP
        then
          NEW_LOC(2) := NEW_LOC(2) + 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_E;

procedure CHECK_W (N_ARRAY : in out NODE_ARRAY;
                   ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(2) > 1 then
          NEW_LOC(2) := NEW_LOC(2) - 1;
          CK_STATE (NEW_LOC, N_ARRAY, CARD_COST, ROOT);
        end if;
end CHECK_W;

procedure CHECK_UP_W (N_ARRAY : in out NODE_ARRAY;
                      ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;
```

```
begin
        IF NEW_LOC(2) > 1 and NEW_LOC(3) > 1 then
          NEW_LOC(2) := NEW_LOC(2) - 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_W;

procedure CHECK_DOWN_W (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(2) > 1 and NEW_LOC(3) < MAX_DEP then
          NEW_LOC(2) := NEW_LOC(2) - 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_W;

procedure CHECK_S (N_ARRAY : in out NODE_ARRAY;
                   ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          CK_STATE (NEW_LOC, N_ARRAY, CARD_COST, ROOT);
        end if;
end CHECK_S;

procedure CHECK_UP_S (N_ARRAY : in out NODE_ARRAY;
                      ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(3) > 1 then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
```

```
        end if;
end CHECK_UP_S;

procedure CHECK_DOWN_S (N_ARRAY : in out NODE_ARRAY;
                        ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(3) < MAX_DEP
        then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_S;

procedure CHECK_SE (N_ARRAY : in out NODE_ARRAY;
                    ROOT    : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) < MAX_COL
        then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) + 1;
          if ROOT.LOC(4) = 2 then
                NEW_LOC(4) := 3;
          else
                NEW_LOC(4) := 2;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_SE;

procedure CHECK_UP_SE (N_ARRAY  : in out NODE_ARRAY;
                       ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) < MAX_COL
```

```
        and NEW_LOC(3) > 1 then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) + 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          if ROOT.LOC(4) = 2 then
                NEW_LOC(4) := 3;
          else
                NEW_LOC(4) := 2;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_SE;

procedure CHECK_DOWN_SE (N_ARRAY  : in out NODE_ARRAY;
                         ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) < MAX_COL
        and NEW_LOC(3) < MAX_DEP then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) + 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          if ROOT.LOC(4) = 2 then
                NEW_LOC(4) := 3;
          else
                NEW_LOC(4) := 2;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_SE;

procedure CHECK_SW (N_ARRAY  : in out NODE_ARRAY;
                    ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) > 1 then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) - 1;
          if ROOT.LOC(4) = 3 then
```

```
                NEW_LOC(4) := 4;
          else
                NEW_LOC(4) := 3;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_SW;

procedure CHECK_UP_SW (N_ARRAY  : in out NODE_ARRAY;
                       ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) > 1 and
        NEW_LOC(3) > 1 then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) - 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          if ROOT.LOC(4) = 3 then
                NEW_LOC(4) := 4;
          else
                NEW_LOC(4) := 3;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_SW;

procedure CHECK_DOWN_SW (N_ARRAY  : in out NODE_ARRAY;
                         ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) < MAX_ROW and NEW_LOC(2) > 1 and
        NEW_LOC(3) < MAX_DEP then
          NEW_LOC(1) := NEW_LOC(1) + 1;
          NEW_LOC(2) := NEW_LOC(2) - 1;
          NEW_LOC(3) := NEW_LOC(3) + 1;
          if ROOT.LOC(4) = 3 then
                NEW_LOC(4) := 4;
          else
                NEW_LOC(4) := 3;
```

```
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_DOWN_SW;

procedure CHECK_NW (N_ARRAY  : in out NODE_ARRAY;
                    ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) > 1 and NEW_LOC(2) > 1 then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          NEW_LOC(2) := NEW_LOC(2) - 1;
          if ROOT.LOC(4) = 1 then
                 NEW_LOC(4) := 4;
          else
                 NEW_LOC(4) := 1;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_NW;

procedure CHECK_UP_NW (N_ARRAY  : in out NODE_ARRAY;
                       ROOT     : in out LIST_PTR) is

        NEW_LOC : LOC_ARRAY := ROOT.LOC;

begin
        IF NEW_LOC(1) > 1 and NEW_LOC(2) > 1 and
        NEW_LOC(3) > 1 then
          NEW_LOC(1) := NEW_LOC(1) - 1;
          NEW_LOC(2) := NEW_LOC(2) - 1;
          NEW_LOC(3) := NEW_LOC(3) - 1;
          if ROOT.LOC(4) = 1 then
                 NEW_LOC(4) := 4;
          else
                 NEW_LOC(4) := 1;
          end if;
          CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
        end if;
end CHECK_UP_NW;
```

```
    procedure CHECK_DOWN_NW (N_ARRAY  : in out NODE_ARRAY;
                             ROOT     : in out LIST_PTR) is

          NEW_LOC : LOC_ARRAY := ROOT.LOC;

    begin
          IF NEW_LOC(1) > 1 and NEW_LOC(2) > 1 and NEW_LOC(3)
<
          MAX_DEP then
            NEW_LOC(1) := NEW_LOC(1) - 1;
            NEW_LOC(2) := NEW_LOC(2) - 1;
            NEW_LOC(3) := NEW_LOC(3) + 1;
            if ROOT.LOC(4) = 1 then
                  NEW_LOC(4) := 4;
            else
                  NEW_LOC(4) := 1;
            end if;
            CK_STATE (NEW_LOC, N_ARRAY, DIAG_COST, ROOT);
          end if;
    end CHECK_DOWN_NW;

end THE_MOVE;
```

LIST OF REFERENCES

[Busby and Vadus 90]
James G. Busby and Joseph R. Vadus, "Autonomous Underwater Vehicle R & D Trend *Sea Technology*, Vol. 31, no. 3, pp 66 - 73, May 1990.

[Rogers 89]
Ray Charles Rogers, *A Study of 3-D Visualization and Knowledge-based Mission Planning and Control for the NPS Model 2 Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1989.

[Cloutier 90]
John Cloutier, *Guidance and Control System for an Autonomous Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1990.

[Healey 90]
A. J. Healey, et al, "Mission Planning, Execution,and Data Analysis for the NPS AUV II Autonomous Underwater Vehicle," paper presented at Monterey Bay Area Research Institute Workshop on Mobile Undersea Robotics, October 1990.

[Floyd 91]
Charles Floyd, *Design and Implementation of a Collision Avoidance System for the NPS Autonomous Underwater Vehicle (AUV II) Utilizing Ultrasonic Sensors*, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

[Bihari 90]
T. E. Bihari, "Comments on the Computer Software and Hardware of the AUV-II," unpublished paper presented to the NPS AUV research effort, June 1990.

[Ong 90]
S. M. Ong, *A Mission Planning Expert System with 3D Path Optimization for the NPS Model 2 Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, June 1990.

[Jurewicz 90]
Thomas A. Jurewicz, *A Real-time Autonomous Underwater Vehicle Simulator*, Master's Thesis, Naval Postgraduate School, Monterey, CA, December 1990.

[Bellingham and Consi 90]
James G. Bellingham and Thomas R. Consi, "Robots Underwater - Ongoing Research at MIT Sea Grant," *Sea Technology*, Vol. 31, no. 3, pp. 23 - 29, May 1990.

[Zheng et. al. 90]
X. Zheng, E. Jackson, M. Kao, "Object-oriented Software Architecture for Mission-configurable Robots," Proceedings of the first IARP Workshop on Mobile Robots for Subsea Environments, Monterey, CA, 23 - 26 October 1990.

[Latombe 91]
Jean-Claude Latombe, *Robot Motion Planning*, ch 1 - 3, Kluwer Academic Publishers, 1991.

[Herman 86]
M. Herman, "Fast, Three-dimensional, Collision-free Motion Planning," paper presented at the IEEE International Conference on Robotics and Automation, 1986.

[Kwa 89]
J. B. H. Kwa, "BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm," *Artificial Intelligence*, pp. 95 - 109, February 1989.

[Warren 90]
C. W. Warren, "A Technique for Autonomous Underwater Vehicle Route Planning," IEEE 1987, pp. 199 - 204, 1987.

[Lozano-Perez 83]
Tomas Lozano-Perez, "Spatial Planning: A Configuration Space Approach," IEEE 1983, pp. 109 - 119, 1983.

[Mcghee 90]
R. E. Mcghee, "Two Dimensional Tendril Search," class notes presented at the Naval Postgraduate School, Monterey, CA, 1990.

[Bonsignore 90]
J. Bonsignore, Jr., "Four Dimensional Tendril Search," class presentation at the Naval Postgraduate School, Monterey, CA, 1990.

[Voltz, et al 84]
R. A. Volz, et al, "CAD, Robot Programming and Ada," NATO ASI Series, Robotics and Artificial Intelligence, Vol. F11, pp. 237 - 246.

[Gaffney 89]
Software Productivity Consortium, SW_REUSE_ECONOM-89040-N, An Economics Foundation for Software Reuse, by John E. Gaffney, Jr., July 1989.

[Korf 88]
Richard E. Korf, "Real-time Heuristic Search: New Results*," *Automated Reasoning*, pp. 149 - 144.

[Magrino 91]
Chris Magrino, *Three Dimensional Guidance for the NPS Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

BIBLIOGRAPHY

1. Bennett, S. and Linkens, D. A., *Real-Time Computer Control*, Peter Peregrinus Ltd., London, U.K., 1984.

2. Bonsignore, Jr., J., "Introduction to Configuration Space," class presentation at the Naval Postgraduate School, Monterey, CA, May 1991.

3. Bonsignore, Jr., J., "Real-time Expert Systems," class presentation at the Naval Postgraduate School, Monterey, CA, November 1990.

4. Booch, Grady, *Software Engineering with Ada*, Second Edition, The Benjamin/ Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

5. Byrnes, Ron, "A C Implementation of Tendril Search," class presentation at the Naval Postgraduate School, Monterey, CA, Sepetember 1990.

6. Caddell, Tymothy W., "3 Dimensional Path Planning Using a Tangential Method," presentation at the Naval Postgraduate School, Monterey, CA, 1990.

7. Compton, Mark A., "Robot Simulator for NPS AUV Track and Obstacle Avoidance Analysis," class presentation at the Naval Postgraduate School, Monterey, CA, May 1991.

8. Fu, R. S., Gonzalez, R. C., Lee, C. S. G., *Robotics Control, Sensing, Vision, and Intelligence*, McGraw-Hill Book Company, New York, NY, 1987.

9. Glass, Robert L., *Real-Time Software*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

10. Healey, Anthony J., "Planning, Navigation, Dynamics and Control of Autonomous Underwater Vehicles," proposal for research, Naval Postgraduate School, October 1990.

11. Hecker, Michael, "Artificial Potential Fields," class presentation at the Naval Post graduate School, Monterey, CA, June 1991.

12. Kanayama, Yutaka and De Haan, Gregory R., "A Mathematical Theory of Safe Path Planning," class notes at the Naval Postgraduate School, Monterey, CA, April 1991.

13. Khatib, Oussama, "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," *The International Journal of Robotics Research*, Vol. 5, No. 1, pp. 90 - 98, Spring 1986.

14. Levi, Shem-tov and Agrowala, A. K., *Real-Time System Design*, McGraw-Hill Publishing Company, New York, NY, 1990.

15. Levitt, Tod S., Lawton, Daryl T., "Qualitative Navigation for Mobile Robots," Artificial Intelligence and International Journal, Vol. 44, No. 3, August 1990.

16. Manber, Udi, *Introduction to Algorithms a Creative Approach*, pp. 204 - 208, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

17. O'Sullivan, Stephani, "Autonomous Control Logic Industry Brief," presentation made by Stephanic O'Sullivan, presentation slides, Panama City, FL, October 1989.

18. Richbourg, R. F., Rowe, Neil C., Zyda, Michael J., McGhee, R. B., "Solving Global, Two-Dimensional Routing Problems using Snell's Law and A* Search," *IEEE International Conference on Robotics and Autonomous Vehicles*, Vol. 3, 1987.

19. Sharma, D.D. and Sridharan, "Knowledge-Based Real-Time Control: A Parallel Processing Perspective," *Machine Architectures and Computer Languages for AI*, 1987.

20. Skansholm, Jan, *Ada From the Beginning*, Addison-Wesley Publishing Company, Woking England, 1988.

21. Software Productivity Consortium, *Tools for Static Analysis of Ada Source Code*, Software Productivity Consortium, Herndon, VA, June 1990.

22. Wilkinson, Paul, "High Level Control of an AUV," class presentation at the Naval Postgraduate School, Monterey, CA, May 1990.

23. Wilkinson, Paul, "Tendril Search Path Planning/Path Replanning with Dynamic Obstacles," class presentation at the Naval Postgraduate School, Monterey, CA, September 1990.

24. Yourdon, Edward, *Modern Structured Analysis*, Yourdon Press, Englewood Cliffs, NJ, 1989.

25. Zyda, Michael J., et al, "Three-Dimensional Visualization of Mission Planning and Control for the NPS Autonomous Underwater Vehicle," *IEEE Journal of Oceanic Engineering*, Vol. 15, no. 3, pp. 217 - 221, July 1990.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943	2
Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
Commandant of the Marine Corps Code TE 06 Headquarters, U.S. Marine Corps Washington, D.C. 20380-0001	1
Gleen Reid, Code U401 Naval Surface Warfare Center Silver Spring, MD 20901	1
RADM Evans, Code SEA92 Naval Sea Systems Command Washington, D.C. 20362	1
Dr. G. Dobeck, Code 4210 Naval Coastal Systems Center Panama City, FL 32407-5000	1
Dick Blidberg Marine Systems Engineering Lab SERB Building 242 University of New Hampshire Durham, NH 03824	1

Mack O'Brien 1
Charles Stark Draber Laboratory, Inc.
Mail Station 5C
555 Technology Square
Cambridge, MA 02139

Dr. Dana Yoerger 1
Woods Hole Oceanographic Institute
Woods Hole, MA 02543

Dr. Yuh-jeng Lee 2
Computer Science Department code CS/LE
Naval Postgraduate School
Monterey, CA 93943